Mälardalen University Press Doctoral Theses
No.121

Managing Extra-Functional Properties in Component-Based Development of Embedded Systems

Séverine Sentilles

June 2012

MÄLARDALEN UNIVERSITY

School of Innovation, Design and Engineering

Copyright © Séverine Sentilles, 2012
ISSN 1651-4238
ISBN 978-91-7485-067-3
Printed by Mälardalen University, Västerås,Sweden

To the ones I love who have always been there for me
when I needed it the most.

Abstract

The continuously increasing complexity of embedded systems is a major issue for their development, which, in addition, must also consider specific extra-functional requirements and constraints, such as limited and shared resources, distribution, timing, and dependability. Thus, embedded systems call for development solutions that can efficiently and predictably cope with these issues. Component-based software engineering is a proven paradigm to handle complexity. Yet, for efficiently managing extra-functional properties, a component model needs to have dedicated mechanisms that provide a suitable support for their management. The objective of this thesis is to build this support.

We have performed a systematic analysis of existing component models and identified challenges of applying a component-based approach to embedded system development. Based on these challenges we have advanced the current state-of-the-art by developing a new component model, called ProCom, that accommodates the specifics of embedded systems through its well-defined execution semantics and layered structure. Centered around ProCom, we have also developed PRIDE, the ProCom Integrated Development Environment. PRIDE supports the development from early specification to synthesis and deployment, providing the means to aggregate various analysis and verification tools.

The main contribution of the thesis is in the design and implementation of an extra-functional property management framework that enables to seamlessly specify, manage and integrate multi-valued context-aware extra-functional properties of component-based embedded systems. Properties can be attached to architectural elements of component models and their values can be compared and refined during the development process. In particular, having multiple context-aware values allows values from different sources to be compared. The proposed concepts have been demonstrated on several representative example systems.

Résumé — Abstract in French

L'accroissement continu de la complexité des systèmes embarqués pose un problème majeur pour leur développement lequel doit aussi prendre en compte les exigences extra-fonctionnelles et les contraintes du domaine telles que la limitation et le partage des ressources, la distribution, et les contraintes temporelles et de fiabilité. De ce fait, les systèmes embarqués requièrent de nouvelles solutions pouvant efficacement et de manière prévisible répondre à l'ensemble de ces besoins. L'ingénierie logicielle basée composants est un paradigme qui a déjà démontré des aptitudes pour appréhender la complexité logicielle. Cependant, pour supporter de manière efficace les propriétés extra-fonctionnelles, un modèle de composants doit posséder des mécanismes spécifiques. L'objectif de cette thèse est de construire un tel support.

Pour ce faire, nous avons analysé de manière systématique des modèles de composants existants à ce jour et identifié des challenges relatifs à la réalisation d'une approche basée composants dédiée au développement des systèmes embarqués. S'appuyant sur ces challenges, nous avons avancé l'état de l'art en développant ProCom, un nouveau modèle de composants qui répond aux attentes des systèmes embarqués au travers de sa sémantique d'exécution et de sa structuration en niveaux. Centré autour de ProCom, nous avons aussi développé PRIDE, son environnement de développement intégré. PRIDE couvre le procédé de développent des premières phases de spécification jusqu'à la synthèse et le déploiement et fournit des moyens d'intégrer différents outils d'analyse et de vérification.

La contribution principale de cette thèse réside dans la modélisation et la réalisation d'un support pour la gestion des propriétés extra-fonctionnelles pour les systèmes embarqués construits à base de composants logiciels. Ce

support facilite la spécification, le management et l'intégration de propriétés multivaluées tenant compte du contexte dans lequel elles ont été établies. Les propriétés peuvent être attachées aux éléments architecturaux des modèles de composants et leurs valeurs peuvent être comparées et raffinées durant le développement. En particulier, le fait d'avoir des valeurs multiples avec leur contexte d'évaluation permet de comparer des valeurs provenant de différentes sources. Les concepts proposés ont été illustrés au travers d'exemples représentatifs de systèmes.

Acknowledgements

When I started my Ph.D. studies, I heard many people saying that getting a Ph.D. is a journey. Freshly graduated, I could not really understand how much different from getting a Master degree it was. But when I look back at it, now that I am about to finish, I see what they meant and, of course, they were right! This is a journey! A journey with its good and bad, its unanticipated events and challenges, a lot of travels (way more than what I was expecting :)) and plenty of amazing experiences. To me, it has been an adventure that I am really happy to have set off for. But this adventure would not have been possible nor enjoyable if I had to go through it alone. And, as the journey ends, I take the opportunity to express my deepest thanks to all who have contributed to make it so great for me.

My first thanks go to the ones without whom I would never have started my graduate studies here at Mälardalen University. I owe a big part of this to Nicolas Belloir who put his trust in me and always tried to pushed me forward, smoothly enough to make me apply to a PhD position here at MDH and accept it! And, of course, involved in this are my supervisors, Ivica Crnković and Hans Hansson. Thank you so much for believing in me and accepting me as a PhD student. I am always amazed by your enthusiasm, commitment and above all your inexplicable capacity to work so much, especially when it is for others! Also, many thanks go to my assistant supervisor, Jan Carlson, for the fruitful discussions, inputs, reviews, help and guidance every time I needed it, also for always finding nice ways to give comments. I also want to thank my French supervisors, Franck Barbier and Eric Cariou, who have given me the opportunity to do a so-called "co-tutelle" with the university of Pau.

Many thanks are also way overdue to the "Mental Department" and more for contributing to making the department a fun, warm, welcoming and friendly place: Cristina, Svetlana, Bob, Hüs, Tibi, Aida, Adnan, Aneta, Juraj, Luis, Farhang, Hongyu, Andreas (G., H., J.), Leo, Mikael, Eduard, Raluca, Mehrdad,

Federico, Rafia, Saad, Luka, Josip, Jagadish, Batu, Seanna, Fredrik and Moris (+1 ;). Thank you guys for all the laughters and great moments during the fika, lunches and travels. You are really great people to work with, and above all great friends. And of course, I don't forget all the colleagues who also contribute a lot to make IDT's working atmosphere so pleasant: Paul, Sasi, Radu, Daniel, Gordana, Stefan, Sigrid, Barbara, Jan G., Björn, Kristina, Mic, Hang, Jiale, Damir, Lars, Anton, Rikard, Stig, Frank, Jukka, Thomas, Antonio, Malin (R., Å.), Gunnar, Åsa, Carola and Suzanne.

There are also lots of friends from childhood and university that I really want to thank for having been present for me when I really needed support and good friends, and this despite being geographically quite far away: Anouk, Flo, Natacha, Aurel, Cristine, Fafou, Eric, Gael, Sophie, Marie, Pauline, Laure, Aude, Anne-Sophie and Bea. I must say that I am really lucky to have so good friends around.

And last but not least, I would like to thanks my number one supporters: my family. I have no word to express how much I owe you for always being there for me and supporting me no matter what! It is a strength for me to know that I can always count on you. Finally, my last thoughts are for two persons, my mum and grandad, who always pushed me to do my best in everything I tried. I wish they were still here today and I hope that from above, they can see this now and are proud of me. I really wish that my mum could tell me today as she always did in the past *"Bon, t'aurais pu faire mieux quand même!"* with her usual loving smile.

Séverine Sentilles
Västerås, June 2012

This work has been supported by the Swedish Foundation for Strategic Research (SSF), via the research centre PROGRESS.

List of Publications

Key Publications Related to the Thesis

Paper A: *A Classification Framework for Software Component Models* . Ivica Crnković, Séverine Sentilles, Aneta Vulgarakis, Michel Chaudron. In IEEE Transaction of Software Engineering, vol 37, nr 5, p593-615, October, 2011.

Paper B: *A Component Model Family for Vehicular Embedded Systems*. Tomáš Bureš, Jan Carlson, Séverine Sentilles, Aneta Vulgarakis. In Proceedings of the 3rd International Conference on Software Engineering Advances (ICSEA), Sliema, Malta, October 2008.

Paper C: *A Component Model for Control-Intensive Distributed Embedded Systems*. Séverine Sentilles, Aneta Vulgarakis, Tomáš Bureš, Jan Carlson, Ivica Crnković. In Proceedings of the 11th International Symposium on Component Based Software Engineering (CBSE), Karlsruhe, Germany, October, 2008.

Paper D: *Save-IDE – A Tool for Design, Analysis and Implementation of Component-Based Embedded Systems*. Séverine Sentilles, Anders Pettersson, Dag Nyström, Thomas Nolte, Paul Pettersson, Ivica Crnković. In Proceedings of the 31st International Conference on Software Engineering (ICSE), Vancouver, Canada, May 2009.

Paper E: PRIDE– *An Environment for Developing Distributed Real-Time Embedded Systems*. Etienne Borde, Jan Carlson, Juraj Feljan, Luka Lednicki, Thomas Lévêque, Josip Maras, Ana Petricic, Séverine Sentilles. In Proceedings of the 9th Working IEEE/IFIP Conference on Software Architecture (WICSA), Boulder, Colorado, USA , June, 2011.

Paper F: *Integration of Extra-Functional Properties in Component Models*. Séverine Sentilles, Petr Štěpán, Jan Carlson and Ivica Crnković. In Proceedings of the 12th International Symposium on Component Based Software Engineering (CBSE), East Stroudsburg University, Pennsylvania, USA, June, 2009.

Paper G: *Integrating Behavioral Descriptions into a Component Model for Embedded Systems*. Aneta Vulgarakis, Séverine Sentilles, Jan Carlson, Cristina Seceleanu. In Proceedings of the 36th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), p 113-118, IEEE, Lille, France, September, 2010.

Paper H: *Refining Extra-Functional Property Values in Hierarchical Component Models*. Thomas Lévêque, Séverine Sentilles. In Proceedings of the 14th International Symposium on Component Based Software Engineering (CBSE), Boulder, Colorado, USA, June, 2011.

Thesis: *Towards Efficient Component-Based Software Development of Distributed Embedded Systems*. Séverine Sentilles. Licentiate Thesis, Mälardalen University, Västerås,Sweden, November, 2009.

Additional Publications Related to the Thesis

- *Flexible Semantic-Preserving Flattening of Hierarchical Component Models*, Thomas Lévêque, Jan Carlson, Séverine Sentilles, Etienne Borde, In Proceedings of the 37th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA), IEEE Computer Society, Oulu, Finland, August, 2011.

- *Evolution Management of Extra-Functional Properties in Component-Based Embedded Systems*, Antonio Cicchetti, Federico Ciccozzi, Thomas Lévêque, Séverine Sentilles, In Proceedings of the 14th International ACM SIGSOFT Symposium on Component Based Software Engineering (CBSE), ACM SIGSOFT, Boulder, Colorado (USA), June, 2011.

- PRIDE , Ivica Crnković, Séverine Sentilles, Thomas Lévêque, Mario Zagar (University of Zagreb), Ana Petricic, Juraj Feljan, Luka Lednicki, Josip Maras, DICES workshop @ SoftCOM 2010, Bol, Croatia, September, 2010.

- *Save-IDE — Integrated Development Environment for Building Predictable Component-Based Embedded Systems*. Séverine Sentilles, John Håkansson, Paul Pettersson, Ivica Crnković. In Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE), L'Aquila, Italy, September 2008.

Other Publications

Conferences and Workshops:

- *Energy Management in Embedded Systems — Towards a Taxonomy*, Umesh Balaji Kothandapani Ramesh, Séverine Sentilles, Ivica Crnković. In Proceedings of the 1st International Workshop on Green and Sustainable Software (GREENS) at International Conference on Software Engineering (ICSE), Zurich, Switzerland, June, 2012

- *Collaboration between Industry and Research for the Introduction of Model-Driven Software Engineering in a Master Program*. Séverine Sentilles, Florian Noyrit, Ivica Crnković. In Proceedings of the Educator Symposium of the ACM/IEEE 11th International Conference on Model Driven Engineering Languages and Systems (MODELS), Toulouse, France, September 2008.

- *Valentine: a Dynamic and Adaptive Operating System for Wireless Sensor Networks*. Natacha Hoang, Nicolas Belloir, Cong-Duc Pham, Séverine Sentilles. In Proceedings of the 1st IEEE International Workshop on Component-based design Of Resource-Constrained Systems (CORCS), Turku, Finland, July 28 - August 1, 2008.

- *A Model-Based Framework for Designing Embedded Real-Time Systems*. Séverine Sentilles, Aneta Vulgarakis, Ivica Crnković. In the Proceedings of the Work-In-Progress (WIP) track of the 19th Euromicro Conference on Real-Time Systems (ECRTS), Pisa, Italy, July 2007.

MRTC reports:

- *Connecting ProCom and REMES*, Aneta Vulgarakis, Séverine Sentilles, Jan Carlson, Cristina Seceleanu, MRTC report ISSN 1404-3041 ISRN MDH-MRTC-244/2010-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, May, 2010.

- *ProCom – the Progress Component Model Reference Manual, version 1.0*. Tomáš Bureš, Jan Carlson, Ivica Crnković, Séverine Sentilles, Aneta Vulgarakis. MRTC report ISSN 1404-3041 ISRN MDH-MRTC-230/2008-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, June 2008.

- *Towards Component Modelling of Embedded Systems in the Vehicular Domain*. Tomáš Bureš, Jan Carlson, Séverine Sentilles, Aneta Vulgarakis. MRTC report ISSN 1404-3041 ISRN MDH-MRTC-226/2008-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, April 2008.

- *Progress Component Model Reference Manual - version 0.5*. Tomáš Bureš, Jan Carlson, Ivica Crnković, Séverine Sentilles, Aneta Vulgarakis. MRTC report ISSN 1404-3041 ISRN MDH-MRTC-225/2008-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, April 2008.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Objectives 5

1.3 Research Questions 6

1.4 Thesis Contributions 7

1.5 Research Method 12

1.6 Thesis Outline 14

2 Classifying Software Component Models **17**

2.1 Main Concepts of Component Models 18

2.2 The Classification Framework 21

2.2.1 Lifecycle 21

2.2.2 Construction 25

2.2.3 Extra-Functional Properties 33

2.2.4 The Classification Overview 39

2.3 Surveying Existing Component Models 40

2.3.1 Component Model Selection 41

2.3.2 Methodology 44

2.4 The Comparison Framework 44

2.4.1 Lifecycle Classification 44

2.4.2 Construction Classification 48

2.4.3 Extra-Functional Properties Classification 53

2.4.4 Component Models and Domains 55

2.5 Conclusions 57

3 Defining Multi-Valued Context-Aware Extra-Functional Properties 59
3.1 Extra-Functional Properties in Component-Based Development 60
3.1.1 An Heterogeneous Data Set 60
3.1.2 Extra-Functional Property and Multi-Valuation 61
3.1.3 Extra-Functional Properties and Reusability 63
3.1.4 Extra-Functional Properties in Hierarchical Component Models 65
3.1.5 Extra-Functional Properties and Component Types and Instances 66
3.2 Definitions 69
3.2.1 Attribute Type 69
3.2.2 Attribute Registry 72
3.2.3 Metadata Type 74
3.2.4 Attribute Instance 76
3.3 Summary and Discussions 78

4 Managing Multi-Valued Context-Aware Extra-Functional Properties 81
4.1 The Inherent Challenges 82
4.2 Identified Supporting Mechanisms per Management Concerns 83
4.3 Two Supporting Mechanisms 87
4.3.1 Value Selection 87
4.3.2 Value Refinement between Component Type and Instances 92
4.4 Summary 96

5 nLight — The Attribute Framework 97
5.1 Overview 98
5.2 Introducing Attributes 99
5.3 Extending Component Models with Attributes 100
5.4 The Registry 101
5.4.1 Specifying Attribute Categories 102
5.4.2 Specifying Attribute Types 103
5.4.3 Specifying Metadata Types 108
5.5 The Graphical User Interface 111
5.6 Summary 112

6 The ProCom Component Model 113
6.1 Domain Requirements for Component-Based Development of Embedded Systems . . . 114
6.1.1 Levels of Abstraction . . . 114
6.1.2 Component Granularity . . . 115
6.1.3 Component vs. System Development . . . 116
6.1.4 Underlying Component Model . . . 117
6.2 A Two-Layer Component Model . . . 119
6.2.1 ProSys — the Upper Layer . . . 120
6.2.2 ProSave — the Lower Layer . . . 122
6.2.3 Integrating the Layers — Combining ProSave and ProSys 126
6.3 Extra-Functional Properties in ProCom . . . 127
6.4 Summary . . . 129

7 PRIDE: The ProCom Integrated Development Environment 131
7.1 Feedbacks from an Initial Prototype . . . 132
7.1.1 Intended Software Development Process . . . 133
7.1.2 SaveIDE — the Save Integrated Development Environment . . . 134
7.1.3 Lessons Learned . . . 139
7.2 Concepts behind PRIDE . . . 142
7.3 Overview of PRIDE . . . 143
7.4 Summary . . . 146

8 Extended Examples 149
8.1 The Turntable . . . 149
8.1.1 Overall System Description . . . 150
8.1.2 Architecting the Turntable in ProCom . . . 151
8.1.3 Attribute Type Identification and Specification . . . 153
8.1.4 Early Formal Analysis . . . 154
8.1.5 Attribute Instance Creation . . . 157
8.2 The Personal Navigation Assistant System . . . 159
8.2.1 Overall System Description . . . 159
8.2.2 Architecting the PNA in ProCom . . . 160
8.2.3 Attribute Type Specification . . . 162
8.2.4 Application on the GPS receiver . . . 164
8.3 The Automatic Driving System . . . 166
8.3.1 Overall System Description . . . 166
8.3.2 Attribute and Metadata Type Specification . . . 171

8.3.3 Developing the Drive-by-Wire System (Iteration 1) . . 174
8.3.4 Enhancing the Drive-By-Wire System with an Automatic Driving Functionality (Iteration 2) 181
8.4 Summary 190

9 Related Work 193
9.1 On Component Model Classification Frameworks 193
9.2 On Extra-Functional Properties 195
9.2.1 Contract-Oriented Approaches 197
9.2.2 Prediction-Oriented Approaches 199
9.2.3 Fact-Oriented Approaches 202
9.3 On Embedded System Development 204
9.3.1 Component Models 204
9.3.2 Alternative Approaches 207

10 Conclusions and Future Work 209
10.1 Summary 209
10.2 Discussions 211
10.3 Future Work 216

Bibliography 219

Chapter 1

Introduction

Development of embedded systems is a complex process subject to several challenges: *i*) complex functionality, *ii*) efficiency of development, *iii*) quality and dependability, and *iv*) specific requirements such as constrained resources or real-time issues. This is the main focus of this thesis, which investigates and proposes methods and techniques to improve software development by helping guaranteeing that the delivered products will meet stringent quality requirements.

1.1 Motivation

A suitable and efficient development process is essential when developing safety-critical systems for a variety of domains such as vehicular, automation, telecommunication and healthcare. A malfunction of these systems may have severe consequences ranging from financial losses (e.g. costs for recall of non-conformity products) to more harmful effects (e.g. injuries to users or in the most extreme cases human's casualties). Along with their traditional mechanical functionality, e.g. a combustion engine or mechanical brakes in a car, these products also contain increasingly more software functionality, such as an anti-lock braking system or an electronic stability control unit in a car.

Functionality in those types of product are provided through special-purpose built-in computers, called *embedded systems*, which are tailored to perform a specific task by a combination of software and hardware. Embedded systems have spread rapidly over the past few decades to be virtually in

any kind of modern appliances such as digital watches, set-top boxes, mp3-players, washing-machines, mobile telephones, cars, aircrafts, forest machines and many others. It is worth noting that the great diversity of devices containing embedded systems makes the boundaries between, what is considered to be embedded systems and what is not, particularly unclear. Many devices share characteristics with embedded systems without necessarily been considered as such. Notebooks, laptops or personal digital assistants are few examples of devices in the grey zone of the definition of embedded systems: they are resources-constrained and possibly integrated into the real world through various equipment such as GPS but they are still regarded as "bigger" than archetypical embedded systems. Conversely although containing desktop-like software and means to interact with users, others devices such as control-system for robots are still considered as embedded systems. Because of this, a uniform definition covering this diversity is difficult to pinpoint and there is currently no unique definition of what they are.

The close interconnection of embedded systems with their surrounding environment and their ability to directly impact on this environment lead to a characteristic shared by many of them: their dependability nature. As defined by Laprie in [1], dependability of a system is the quality of the delivered service such that a user can justifiably rely on this service. In particular, dependability is expressed in terms of safety (i.e. the failure of the system must be harmless), maintainability (probability that a failure can be fixed within a predefined amount of time), reliability (probability that the system will not failed) and availability (probability that the system is working and accessible) among others. This means that to prevent any malfunction, such systems have to react in precisely defined ways, i.e. be predictable.

In addition, many of these systems also have real-time constraints, which means that they must react correctly to events in a given interval of time. When all the timing requirements must strictly be ensured, embedded systems are called *hard real-time systems* whereas *soft real-time systems* are more flexible towards the timing bounds and can tolerate to occasionally violate them. One popular example to illustrate this strong interdependence between real-time and dependability issue is the one of a car airbag. In case of an accident, the airbag has to inflate suitably at a particular point in time, otherwise it is useless for saving the driver's life.

To summarize, in contrast to general purpose computers, embedded systems are typically:

- reactive systems closely integrated into the environment with which they interact through sensors and actuators, and
- strongly resource-constrained in terms of memory, bandwidth and energy,
- facing dependability and real-time constraints.

Thanks to embedded systems, tremendous opportunities are triggered by the introduction of software functionality, sometimes even completely replacing hardware ones. For example, in the automotive domain, the added-value in high-end models of vehicles is generated mainly by the integration of new electronic features that are intended to optimize the costs of utilization (e.g. lower fuel consumption), or to improve the user's comfort or safety. According to [2] in 2006, 20% of the value of each car was due to embedded electronics. This involves features such as airbag control system, anti-braking system, engine control system, electronic stability control system, global positioning system, door locking system, air-conditioning system and many more. More generally speaking, these features concern control, infotainment (i.e. information and entertainment) and diagnosis systems.

However, introducing many software functionalities also considerably increases complexity. For example, as highlighted by Broy [3], a high-end model of vehicle contains today around thousands of software functions corresponding to around 100 millions lines of software code that are executed through a network of 70 to 100 micro-controllers communicating over several dedicated channels. Such a high complexity leads to the fact that the federated architecture solution of decomposing the required functionalities into subsystems that are realised by dedicated computing units using their own microcontroller does not scale anymore. Instead, there is a need to put several subsystems on one physical unit, which implies that resources must be shared between subsystems. Another aspect of this increasing complexity is distribution, where systems are designed as distributed systems communicating over a dedicated network such as a CAN-bus [4] or a LIN-bus [5] in a vehicle. The interdependence of these concerns together with the need for thorough verification of the system make the development of embedded systems rather difficult and time-demanding. For example, in the automotive domain, whereas car manufacturers strive for low production costs since each car model is manufactured

in large quantities, the biggest costs — up to 40% of the development costs [6] — resides in software and electronics costs.

Accordingly, one major issue in dealing with safety-critical real-time embedded systems is to have efficient solutions to deal with the complexity while ensuring that the system always behaves as expected. Their development must hence support thorough analysis and tests, and push these activities even further compared to what can be found in general in software engineering.

A promising solution for the development of distributed embedded systems lies in the adoption of a Component-Based Development (CBD) approach facilitating the different types of analysis. The CBD approach has the goal to increase efficiency in software development by:

- reusing already existing solution encapsulated in well-defined entities (components);
- building systems by composing entities (both from a functional and extra-functional[1] point of view); and
- clearly separating component development from system development.

Stressing reusability, several features of CBD are of high interest in the development of embedded systems such as complexity management, increased productivity, higher quality, shorter time-to-market and lower maintenance costs. Despite those appealing aspects and its establishment as an acknowledged approach for software development, notably for desktop or business applications [7], CBD still struggles to meet all the challenges faced by embedded system development, and this, even though several approaches currently aim at addressing them. These approaches include AUTOSAR [8], BlueArX [9, 10], Rubus [11], Koala [12] in industry and Pecos [13], SaveCCM [14], ROBOCOP [15] and PECT [16] in research.

For a better acceptance in this domain, the main challenge of CBD is to deal with both complexity and functional requirements on one hand, and on the other hand to deal with the specifics related to embedded systems and their development needs, and in particular managing extra-functional properties. More specifically, this requires to have a systematic approach that homogeneously integrates the various activities and related artefacts involved in the development process.

[1]Extra-functional properties are attributes that define "how" a system performs rather than "what" it does. They are expressed through numerous characteristics and can be found under several equivalent denominations: non-functional properties, quality attributes, attributes, etc. Examples of extra-functional properties important for embedded systems include dependability, timing characteristics, and resources consumption.

1.2 Objectives

The main purpose of this thesis is to determine solutions towards establishing an efficient software development of distributed embedded systems abiding by the principles of component-based development that can ensure the quality of the delivered products. Assuming that the principles advocated in CBD are also applicable for developing distributed embedded systems, this thesis discusses how to suitably accommodate the specifics of "traditional" embedded system development with component-based development and, then how to integrate and manage extra-functional properties in the development to ensure the predictability of the final product. This thesis also focuses on determining the required engineering practices and tools to efficiently support the composition theories which have been proposed.

Formulated as a question, the main challenge that this thesis aims at addressing is the following:

> *How can distributed embedded systems be developed in a predictable and efficient way while using the CBD principles?*

This thesis does not provide a direct answer to this question but focuses on solving parts of this challenge:

1. investigating how to apply component-based development principles to embedded system development,

2. establishing the specific requirements for a dedicated component model, and

3. providing a support to manage extra-functional properties throughout the development.

Concretely, in this thesis, we propose a component-based approach for distributed embedded systems supported by the specification of a dedicated component model. This component model is endowed with suitable characteristics, properties, and features to efficiently support the management of the specific concerns of embedded system domain. Further, a special focus is put on extra-functional properties regarding their integration and management to bridge analysis in the development process. The approach is illustrated through the realisation of an integrated development environment (IDE).

1.3 Research Questions

In this section, we break down the main research challenge into a set of more concrete research questions, which have served as basis to frame and guide the different phases of the work described in this thesis.

Research question 1

> *What characteristics of a component model facilitate software design of distributed embedded systems?*

Through this research question, the purpose is (i) to explore and identify important needs in the development of distributed embedded systems (focusing more specifically on the design phase using a CBD approach), and (ii) to propose a new component model endowed with suitable characteristics, properties and features to provide a solution to these needs.

Research question 2

> *What mechanisms are suitable to support the management of extra-functional properties within a component model?*

In embedded system development, extra-functional properties are as important for system correctness as the functionality itself but more challenging. From the results obtained in answering the previous research questions, it has been observed that, although essential, extra-functional properties are seldom considered in component-based development. In most cases, they are evaluated in late development phases through simulation and/or measurement, which might be costly if the extra-functional requirements are not satisfied. In some cases, extra-functional properties are considered in early development phases only to serve as predictions. Furthermore, few component models provide support for dealing with extra-functional properties, and often, this support addresses a predefined subset of extra-functional properties only.

Following these observations, we formulated the aforementioned research question, which addresses mainly the predictability aspect needed in the development of distributed embedded systems. In that respect, this research question focuses on determining a way to enhance component models to provide

the necessary grounds to efficiently support, in a systematic way, the management of extra-functional properties in a component-based development for embedded systems. Furthermore, through this research question the aim is also investigate solutions to develop a corresponding extra-functional property management framework.

Research question 3

> *How can the different aspects of component-based development for embedded systems be seamlessly integrated into a development environment?*

This research question addresses the needs required to support in practice the development of embedded systems. Given that for embedded system development, both functional and extra-functional correctness must be considered, different techniques must be used all along the development starting from modelling low-level functionality, using a behaviour model to enable early predictions, and/or using test-cases, simulation and measurements. Up till now, the tools implementing these techniques are rather independent and often require manual effort to use them together. Accordingly, one of the important challenge that exist in embedded system development is to find a way to provide easy and tight integration of the various techniques and tools required for the development of distributed embedded systems. Hence, the main goal with this research question is to attempt to develop a prototype that can be used as a basis to both demonstrate the feasibility of the proposed ideas and evaluate their advantages and drawbacks in using them in practice.

1.4 Thesis Contributions

The thesis provides the following four main contributions:

1. A classification framework for component models;
2. A general framework for the management of extra-functional properties in component-based development;
3. A new component model for control-intensive embedded systems;
4. Two Integrated Development Environments for component-based embedded systems.

For each contribution a summary, the relation to the thesis and my personal involvement in its realization are detailed below. These contributions are the outcome of a set of results that address the main research challenge and questions presented in the previous sections. While studying the current state-of-the-art of component based software engineering and providing a classification of the characteristics of component models, the existence of a large variety of component models has been identified — some of them targeting embedded systems specifically. This has led to contribution 1, which in its turns exposed the lack of appropriate management support for extra-functional properties in component models. From this contribution, we also identified some common features among existing component models for embedded systems. As a result, contributions 2 and 3 were devised before being integrated together through the corresponding implementation of the attribute framework for contribution 3 and integrated development environments (contribution 4) for contribution 2. Benefiting from these implementations, we realised several examples which outcomes have had influenced the previous contributions. These relations are illustrated in Figure 1.1 together with the correspondence between the thesis contributions and the chapters of the thesis.

1. **A classification framework for component models**

   This contribution introduces a systematic classification of characteristics of component models. It relies on a thorough study of twenty-four component models to discuss basic principles of component models and component-based software engineering and identify common characteristics of component. From this, a Component Model Classification Framework is proposed and used to classify the twenty-four component models. In analyzing the classified component models, it is possible to pinpoint differences and identifies characteristics shared by component models developed for a similar domain, such as embedded systems. Paper A [17] has been published as the main outcome of this contribution and is used as a basis for Chapters 2 and 9.

   Personal contributions:
   I personally contributed to this work with the initial idea of the component model classification, a first simple prototype with few component models and aspects only, and together with Aneta Vulgarakis in collecting, analyzing and classifying additional component models. I had the main responsibility over the construction dimension and the related work section. Everyone worked equally in the iterative process to refine the framework and contributed with discussions, reviews and suggestions.

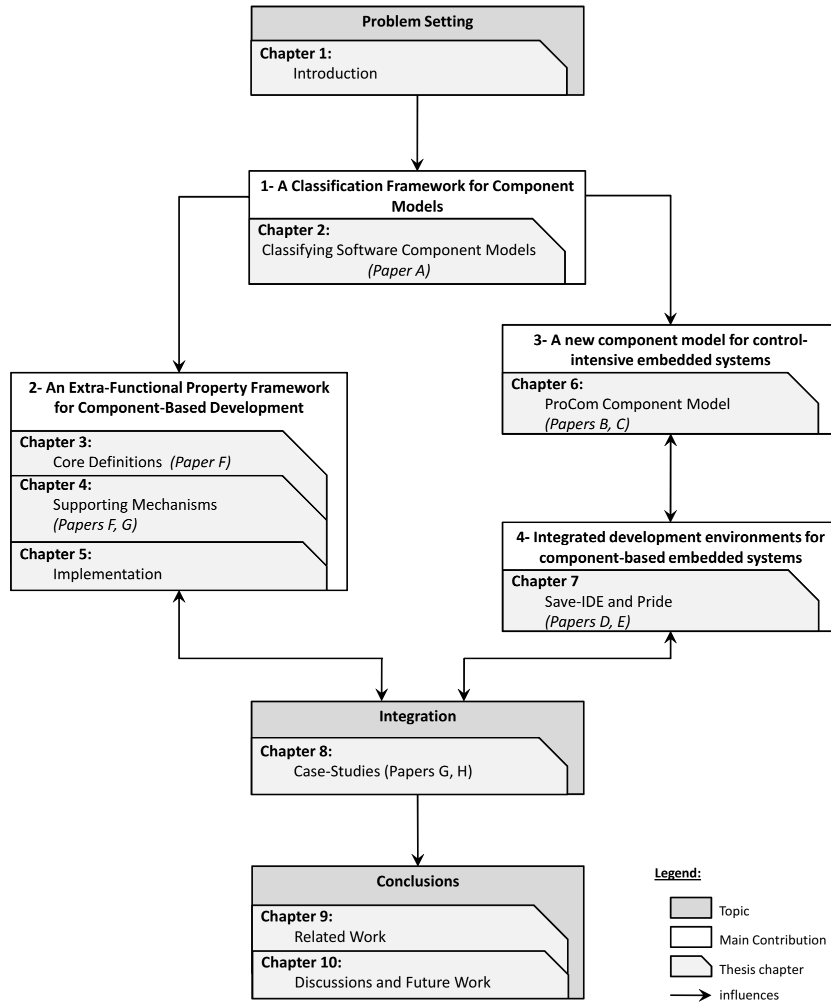


Figure 1.1: Relation between the thesis contributions and the chapters of the thesis.

2. **A general framework for the management of extra-functional properties in component-based development**

   This framework enables the specification of multi-valued and context-aware extra-functional properties and propose a support for their uniform and seamless management in component-based development. Properties can be attached to selected architectural entities of component models. Their values can be compared and refined during the development process. In particular, thanks to having multiple context-aware values, values from different sources can be compared and reused in appropriate context. This is done with the main objective of providing an efficient support, possibly automated, for analysing selected properties. This contribution includes *i*) a study of the possible usage of extra-functional properties in component-based development, *ii*) a specification of multi-valued context-aware extra-functional properties, *iii*) an investigation of the necessary supporting mechanisms for specifying, managing, refining extra-functional properties, and *iv*) the implementation of an extensible prototype for the proposed solutions. This is the core contribution of the thesis and the corresponding results have been published in Papers F [18], G [19] and H [20] and are discussed in Chapters 3, 4 and 5 and 9.

   Personal contributions:
   I was the main driver of this work and contributed in identifying the problem of the lack of systematic support of extra-functional properties during component-based development, in developing the concept of multi-valued context-aware extra-functional properties, and investigating needed supporting mechanisms. I also supervised the realisation of the first prototype implementing the concepts of multi-valued extra-functional properties, prototype that I have refined and enriched later on. Ivica Crnkovic, Jan Carlson and Thomas Lévêque contributed with valuable discussions, feedbacks and ideas.

3. **A new component model for control-intensive embedded systems**

   In this contribution, a component model for the design and development of control-intensive distributed embedded systems called ProCom has been developed. The particularity of ProCom lies in the existence of two layers designed to cope with the different design paradigms which exist on different abstraction levels in distributed embedded systems. Each layer is hierarchical and has its own architectural style and communication paradigm. Moreover, through its restricted semantic ProCom provide a ground for analyzing

the components and predict their properties, such as resource consumption and timing behaviour, already in early development phases. The results from this contribution have been published in Paper B [21] and C [22] and are described in Chapter 6.

Personal contributions:
ProCom is the result of a team work involving many members of the PROGRESS project[2] which I participated in. I personally contributed to this topic by actively participating in the discussions concerning the development process, the discussions with the domain experts to collect information on their needs and by influencing some of the decisions through my parallel work on the realization of an integrated development environment, called Save-IDE, for the SaveCCM component model, which are predecessors of PRIDE and ProCom respectively.

4. **Two Integrated Development Environments for component-based embedded systems**

This contribution provides an extensible development framework to evaluate in practice research contributions centered around the proposed component model and a support to integrate the attribute framework. Two prototypes of integrated development environments to support the proposed component-based development approach for distributed embedded systems have been specified and developed. These prototypes enable having components throughout the development process, from early design to deployment and synthesis, and facilitates the integrations of research ideas. Benefiting from the experience gained from developing the Save-IDE, we have built PRIDE, the ProCom Integrated Development Environment. PRIDE is based on an architecture relying on components with well-defined semantics that serve as the central development entity, and as means to support and aggregate various analysis and verification techniques throughout the development from early specification to synthesis and deployment. PRIDE also provides generic support for integrating extra-functional properties into architectural definitions through the integration of nLight, the framework for the systematic management of extra-functional properties. Results from this contribution have been published in [23], in Paper D[24] and Paper E [25] and are used as basis for Chapter 7.

[2] http://www.mrtc.mdh.se/progress/

Personal contributions:
Concerning the realization of the Save-IDE, I was a member of the developing team with the main responsibility for the design part, including the design of the underlying metamodel and the development of the design tools. This included implementation, testing, bug fixing, working on the final integration, and supervision of master students, etc. For PRIDE, my contributions are derived from my role as the main software architect and include the elicitation of the desired underlying concepts that should guide the development of PRIDE and its design specification together with additional managerial activities for the releases. Concerning the implementation, I was mainly responsible for integrating nLight.

1.5 Research Method

In this thesis, we followed a methodology adapted from the guidelines proposed by Shaw in [26] to perform software engineering research.

This approach starts with the identification of a problem from the real world (*Problem Identification*), in our case the limitations of the current development methods for distributed embedded systems due to the increasing complexity of new embedded system functionalities. The problem is then transferred into a research setting to be investigated with the prospects of findings solutions to it. However, since real world problems are generally quite complex, the scope of the problem needs first to be restricted to be manageable within a research context (*Problem Setting*). This limitation made us focus on a particular aspect of the real problem by formulating the research problem that will be addressed within the work (*Problem Formulation*), and then by stating *Working Assumptions* and *Research Questions*, which together set a frame for the work. Similarly to passing from a real world problem to a research problem, breaking down the research problem into a set of research questions narrows down even further the problem to investigate and helps on focusing on particular aspects of the research problem. In that sense, the working assumptions provide a starting point to the work whereas the research questions correspond more to the specification of the angle of attack chosen to investigate the research problem.

Once the problem is clearly defined, the research work starts with the study of related theories, methods, approaches, techniques or solutions that have already been performed on the topic (*Background Theories*). With the knowledge of the existing state-of-the-art and the questions to answer, some solutions can be devised (*Solutions*). Formulating solutions is not a straightforward process

but an iterative one, in which preliminary ideas are formulated, worked out, refined or even sometimes left aside. When the ideas are mature enough, they must be evaluated and validated to check whether they really answer the research question in a suitable way (*Validation*). If this step fails, the proposed solutions need again to be revisited, refined, improved or thrown away. In that sense, this is an iterative trial and error process, in which analysing the causes of the erroneous solutions might provide useful inputs to find new, better or simply working solutions.

After the validation step is satisfied, the applicability of the proposed solutions to solve the real-world problem can be evaluated (*Evaluation*). An overview of this approach is given in Figure 1.2.

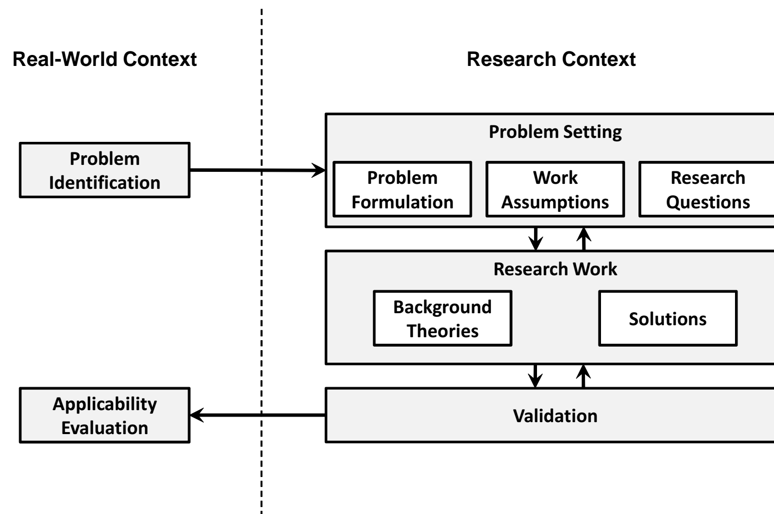


Figure 1.2: Overview of the applied research process.

Each research questions can be answered in different ways and in applying different approaches, thus we describe below the methodology that has been used in the research work described in the previous sections.

To answer research question 1, we proceeded by systematic analysis of existing component models and construction. The process started by studying both the needs in the development process of distributed embedded systems and the current state-of-the-art of component-based software engineering fo-

cusing on existing component models, in particular SaveCCM [14]. This study was based on literature surveys and discussions with domain experts of vehicular and automation domains. Based on these findings, requirements for the component model were extracted and served as foundations in the elaboration of ProCom, which addresses some of the limitations of SaveCCM.

As for the work concerned with research question 2, we used an approach by construction. The work also started with a literature surveys on extra-functional properties and their management and the identification of a few properties of interest in the development process. Then we related their management to their utilisation within the development process. The methodology followed here was iterative and started with the development of a prototype implementing some preliminary ideas to get a better understanding of their integrations and contributions in the development process. From the utilisation of the prototype on development examples, the proposed solutions were refined and additional supporting mechanisms were identified as required.

As for the work concerned with research question 3, we investigated the feasibility of integrating various aspects of component-based approach tightly into a common development environment. Here, we also proceeded by construction. We started by building a prototype of an integrated development environment based on the SaveCCT approach, using the SaveCCM component model and enabling early formal analysis of timing properties. Based on the lessons we learned from building this prototype, we developed a second integrated development environment for ProCom.

1.6 Thesis Outline

The thesis includes the following chapters:

Chapter 1: Introduction This chapter introduces the research setting for the work in detailing the motivation for the work, the research setting and the research questions. Additionally, an overview of the thesis contributions is presented together with the followed research process and research methods.

Chapter 2: Classifying Software Component Models presents a thorough investigation of the concepts related to the notion of component models based on which a classification framework that highlights similitude and differences between twenty-four component models is built.

Chapter 3: Defining Multi-Valued Context-Aware Extra-Functional Properties identifies challenges related to extra-functional properties in component-based development and formally establishes the core definitions supporting the concept of multi-valued context-aware extra-functional properties. These definitions set the basis for the management of extra-functional properties in component-based development.

Chapter 4: Managing Multi-Valued Context-Aware Extra-Functional Properties identifies required supporting mechanisms to handle multi-valued context-aware extra-functional properties within a component-based development. Examples of such mechanisms include filtering, value selection, value comparison and value merging.

Chapter 5: nLight — The Attribute Framework describes a prototype implementation of a framework enabling the systematic management of multi-valued context-aware extra-functional properties. This framework is extensible: new extra-functional properties can be easily added to component models. To do so, it is developed as a set of Eclipse plugins using the Eclipse Modeling Framework.

Chapter 6: The ProCom Component Model identifies first the requirements to adapt the principles of component-based software engineering to fit the specific needs of embedded system development. Based on that, a new component model, called ProCom, dedicated to embedded system development is specified in this chapter.

Chapter 7: PRIDE: The ProCom Integrated Development Environment describes the Integrated Development Environment supporting the concepts presented in Chapter 6.

Chapter 8: Extended Examples exemplifies the contributions on several examples, including a turntable system, a personal navigation assistant system, and an automatic driving system.

Chapter 9: Related Work compares the results of the thesis contributions with similar work related to component model classification, extra-functional properties and component models.

Chapter 10: Conclusion and Future Work discusses the contributions provided in thesis and suggests possible extension of this work.

Chapter 2

Classifying Software Component Models

Due to promising features such as alleviating complexity and shortening of development time, component-based software engineering has become a popular development paradigm. However, there is no consensus on the principles behind component-based software engineering and, as a consequence, many component models have been developed in recent years. Most of these component models focus on some specific points of the paradigm and it is now difficult to have a clear picture of their differences or similarities. The main purpose of this chapter is to:

- Ascertain the main concepts related to the notion of component models to make them clearly understandable.

- Derive a classification framework for component models from these main concepts.

- Analyse existing component models to identify their differences and similitudes and provide an overview of the current state of today component models.

2.1 Main Concepts of Component Models

In order to classify component models, a clear understanding of the main concepts and unique terminology used in component-based software engineering (CBSE) is required. Therefore, we define in this section the concepts related to the notion of component models that are *component model*, *component-based system*, *component* and *binding*.

We use the definition proposed in [27] that defines a component itself relatively to a specific component model. This definition points out that a component model covers multiple facets of the development process, dealing with:
1) rules for the construction of individual components, and
2) rules for the assembly of these components into a system.

Definition:

> *A Component model defines standards for (i) properties that individual components must satisfy, and (ii) methods for composing components.*

In this definition, the term "component properties", is meant to include functional and extra-functional specifications of individual components. The term "composing components" is meant to include mechanisms for component interaction. To explain these terms further, we start from an architectural specification of a component-based system.

A component-based system identifies (i) components, (ii) an underlying platform and (iii) the binding mechanisms, as shown in Fig. 2.1 and presented formally as:

$$\mathbb{CBS} = < \mathbb{P}, \mathbb{C}, \mathbb{B} >$$

Where

$\mathbb{CBS}$ = Component-based system; $\mathbb{P}$ = System platform; $\mathbb{C}$ = A set of components C_i; $\mathbb{B}$ = Set of bindings B_i.

A component is executable[1]. In contrast to arbitrary executable code, a component is formed to interact with other components according to predefined rules. In other words, a component is a software module that includes both execution code and machine-readable metadata (typically including the

[1]Note that executable-property does not necessarily mean binary code. For example. the execution can be achieved through an interpreter or by a virtual machine, or even through compilation before the execution.

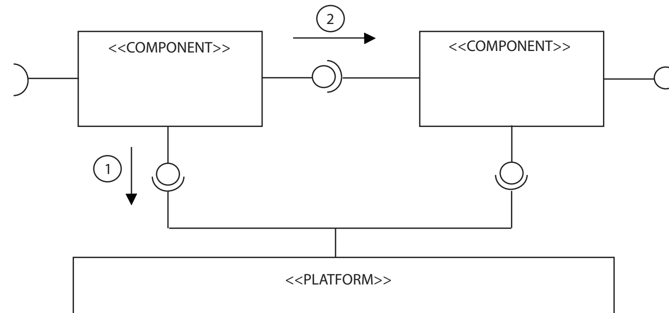


Figure 2.1: Component-based system

interface-signature) which explicitly describes the services that the software provides and the services that it requires from other components and its execution environment. The metadata supports the component framework in composing a component with other components, and in deploying it into an execution environment. In addition, the metadata can include information about extra-functional properties of components.

More formally, we specify a component C by a set of properties. Properties are used in the most general sense as defined by standard dictionaries, e.g.: "a construct whereby objects and individuals can be distinguished" [28]. There is no unique taxonomy of properties, and there exist different property classifications. One commonly used classification is to distinguish functional from extra-functional properties (also designated as non-functional, or Quality of Services, or "ilities"). While functional properties describe functions or services a component provides or requires, extra-functional properties (EFPs) describe its non-functional characteristics. Typical examples of extra-functional properties are quality attributes such as reliability and response-time. A component C can expose its functional properties by the means of an interface I. Hence, we can characterize a component C by its functional interface I and by a set of extra-functional properties P:

$$\begin{aligned} C = \langle I, P \rangle \,,\; with\; I &= \{i_1, i_2, ..i_n\}; \\ P &= \{p_1, p_2, ..p_k\}. \end{aligned}$$

I defines a set of functional properties (services) i_k that a component provides or requires.

P defines a set of extra-functional properties p_i of the component.

If a component $C = \langle I, P \rangle$ complies with a component model CM, then this implies that its interface and its properties must comply with the rules of the component model. This is formally denoted as follows:

$$C \models CM \Rightarrow I, P \models CM$$

Bindings define connections between interfaces. We distinguish bindings between (i) the components and the platform (which enables component integration into a system) from (ii) bindings between components (which enables component interaction). In the first case, we talk about *component deployment* (denoted as ① in Fig. 2.1) and in the second about *component binding* (denoted as ②).

The components C_1 and C_2 bounded by their interfaces I_1 and I_2 construct an *assembly* $A = \{C_1, C_2\}$. If a component model includes assembly as an architectural element, then the assembly is specified by its interface I_A:

$$A = \{C_1, C_2\}, A = \langle I_A \rangle \, | I_A = \langle I_1 \oplus I_2 \rangle$$

Note that an assembly is not necessary a component itself; it is not necessary that it conforms to the component model. If an assembly $C = \{C_1, C_2\}$ conforms to the component model, i.e.

$$C = \langle I, P \rangle \, ; I = \langle I_1 \oplus I_2 \rangle \, , C \models CM$$

the assembly is a component, also called a composite component.

A composite component also exhibits a set of extra-functional properties. In the above example, the composite component is specified by $C = \langle I, P \rangle$ but we did not defined P as a composition of component properties P_1 and P_2. We can state a question: Can P be defined as a composition of P_1 and P_2? As we will see later, the extra-functional properties of a composite component are in most cases not only the result of component property composition, but also of the external environment (e.g. underlying platform and other components). Formally, we express this as

$$C = \langle C_1 \oplus C_2 \rangle \Rightarrow I = \langle I_1 \oplus I_2 \rangle \wedge P_{ex} \vdash P = \langle P_1 \oplus P_2 \rangle$$

where P_{ex} denotes a specification of the external (system) context that has an impact on the composition of component extra-functional properties. A more detailed discussion about binding and composition is presented in Section 2.2.2.

2.2 The Classification Framework

The rules a component model defines for the design and composition of components cover different principles and hide many complex implementation mechanisms. Furthermore, different component models cover different phases in the component lifecycle; while some support only the modelling and design stage, others support mainly the implementation and run-time stages. For this reason, we cannot simply list all possible component models characteristics, but we group the characteristics according to their similar concerns and aspects.

Starting from these premises, we divide the basic characteristics and principles of component models into the following three dimensions:

D.1 **Lifecycle.** The lifecycle dimension identifies the support provided by a component model and the component forms throughout the lifecycle of components. CBSE is characterized by a separation of the development processes of individual components from the development process of the overall system. A component lifecycle covers stages from the component specification until its integration into the systems and possibly its execution and replacement.

D.2 **Construction.** The construction dimension identifies principles and mechanisms for building systems from components including (i) the component functional specification (of which the *interface* is a prominent part), (ii) the means of establishing connections between the components, i.e. *binding*, and the means of intercommunications, i.e. *interactions* between the components.

D.3 **Extra-Functional Properties.** The extra-functional properties dimension identifies the facilities a component model offers for the specifications, management and composition of extra-functional properties.

Below, we discuss these dimensions and introduce their features, i.e. the characteristics of component models.

2.2.1 Lifecycle

An important characteristic of CBSE is the separation of the development process of the overall system from the development processes of individual components [29]. These processes can be completely independent as for example in the development of COTS (Commercial Off-The-Shelf) components and COTS-based systems, up to the point where a component is integrated into a system.

The development of an individual component follows the following stages (see Fig. 2.2): requirements, design, implementation, deployment and execution. During its lifecycle, a component has different forms [30]: initially, a component is represented by a set of requirements, yet during design the same component is represented by a *set of models*. Subsequently, the same component is represented by means of *source code*, complemented by *metadata*. After deployment, the component is integrated in an execution environment. And at run-time, the same component[2] is now represented by *object-code* of the target platform. Optionally, at intermediate stages, a component may be packaged and represented by means of a set of files in a directory or zip-file. Fig. 2.2 shows these successive stages of a component's lifecycle. The lower half of the figure lists the ways in which components may be represented in that particular stage of the lifecycle. In the figure, the requirements and execution stages are depicted with dashed lines to indicate that in these stages components do not necessarily exist as independent units.

Most component models provide support for several stages of the component's lifecycle. Support in the design stage may consist of a dedicated design notation or predefined approach for modelling different aspects of components. For example, the Koala component model [12] has an explicit design notation which includes representations for, amongst others, components, interfaces, and bindings. Other component models dictate the use of state-machines for modelling the behaviour of components. In the implementation stage, a component model typically defines which construction elements should be used for encoding a component in a programming language. Implementation level rules typically include conventions for the naming and structuring of interfaces. The component models that cover several stages often provide a support for transformation between the different component forms; typical examples are transformations from models to code, such as interface specifications to stubs in programming languages. In some cases, the transformation rules can be quite complex, as for example in the domain of real-time systems in which the design units, the components, are transformed into executable units, the real-time tasks.

[2] Actually, an *instance* of this component.

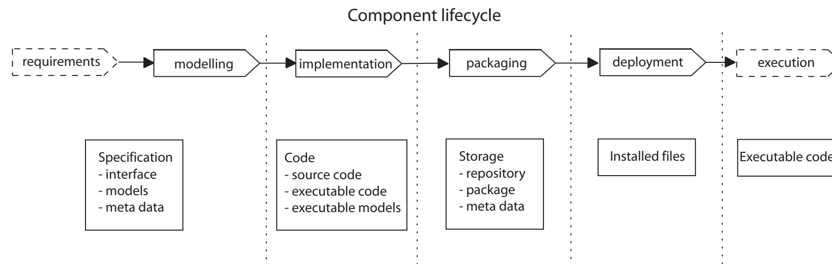


Figure 2.2: Component lifecycle and component forms

Component Lifecycle Stages

We identify the following stages of the component lifecycle:

L.1 *Modelling stage.* Component models provide support for the modelling and the design of components and component-based systems. Models are used either for the architectural description of the systems, the components and the interaction between them (for example using a standard or a dedicated ADL), or for the modelling and verification of particular system and component properties (using different modelling techniques such as statecharts or different variants of finite automata). For example, the KobrA [31] component model uses UML profiles with new or modified UML architectural elements and annotations, while ProCom [22] and Pin [32] have their own modelling languages.

L.2 *Implementation stage.* Component models provide support for the production of code. The support for implementation stage may stop with the provision of the source code, or may continue up to the generation of a binary (executable) code[3]. Most of the component models use standard programming languages. Some component models assume the use of a particular language for the implementation. In such cases, the component model may require that (elements of the) language are used according to some specific rules. For example, the EJB component model [33] uses Java, with some extensions and additional requirements. Others

[3]Considering the component model definition and Szyperski's definition of a component, it can appear strange that component models do not address the implementation stage. However, the component models specify characteristics of components that are executable units, although not necessarily the implementation rules themselves.

component models explicitly aim to be language-independent for the implementation. Such component models may have translators from their modelling and specification languages to a particular, or sometimes multiple, programming language(s) as for CCM [34].

L.3 *Packaging stage.* Because of the separation of the development processes in the component-based lifecycle, there is a need for the storage and packaging of components, either in a repository or for distribution. A component package is a set of metadata and code (source or executable). The metadata contains information about the contents of the files in the package. Accordingly, the result of this stage can be a file, an archive, or a repository in which the packaged components reside prior to their use. For example, in Koala [12], components are packed into a file system-based repository, with one folder per component. The folder includes a number of files: a Component Description Language (CDL) file and, a set of C and header files, test file and different documents. Another example of packaging is used in the EJB [33] component model. There, packaging is done through JAR archives, called EJB-JAR. Each archive contains an XML deployment descriptor, a component description, a component implementation and interfaces.

L.4 *Deployment stage.* At a certain point in time, a component is integrated into an executable system or some target environment, and becomes ready for execution. This may happen at different stages in the system's lifecycle. In general, a component can be deployed at:

(a) *compilation time*: Components are integrated before the system starts executing. Compilation (and linking) achieves integration of components through the resolution of references to interface names. Binding at compilation-time is typical for embedded systems in which the components and the execution platform are compiled and linked together into an executable image. This happens for instance in the Koala component model.

(b) *run-time*: Components may be added or replaced in a system which is executing. Run-time deployment may be realized by using a registry (COM [35]), or by containers which handle installation and communication of the component using information of the deployment descriptor packed with the component implementation (CCM [34], EJB [33]).

2.2.2 Construction

As defined in Oxford advanced learners dictionary [36], *construction* means "the process or method of building". The construction dimension of our classification includes three parts: (i) connection points i.e. *interfaces*, (ii) mechanisms for establishing connections, i.e. *binding* mechanisms, and (iii) communication itself, i.e. *interaction*. The next section discusses each of these aspects in more detail, and provides a list of elements that characterize this dimension.

Interface

A component interface defines a set of actions which is understood by both the provider (the component) and user of that interface (other components, or other software). The actions of an interface can be characterized by a name and a list of parameters that are input to or output from the action. A very common way of specifying an interface is by means of a set of operations (functions) with parameters, as for example used in Java Beans [37] and OSGi [38]. However, there exist other types of interfaces; so called "port-based"[4], where ports are entries for receiving/sending different data types and events, as for instance implemented in IEC61131 [40] and SaveCCM [14]. Fig. 2.3 illustrates the "operation-based" and "port-based" interfaces and interaction styles. In the first case, a component invokes an operation from another component (which may return a result), while in the second case, a component pushes data to another component and possibly starts the execution of this other component by sending a trigger. Alternatively, triggers can be sent by a clock invoking the periodical execution of the component.

Most component models distinguish the actions that components provide to their environment, called *provided interface*, from the actions they require from this environment, called *required interface*. This is an important feature that makes explicit the dependencies of a component. This in turn facilitates independent development and deployment of components.

An interface is not a constituent part of a component, but can exist independently of components as a standard for representing some piece of functionality in a system. The independent existence of interfaces makes it possible to specify interfaces independently of their implementation.

In different stages of development, an interface may be defined through different languages. In the modelling stage, component models may either pro-

[4]Note that the "port-based" concept is different from the concept in UML 2.1 [39] in which a port is defined as a set of interface specifications.

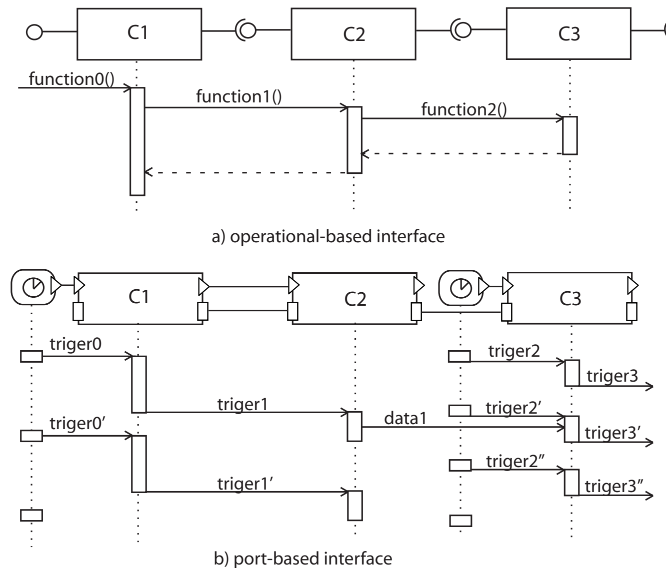


Figure 2.3: Operation-based and port-based interfaces

vide their own languages (often similar to some ADL), or use UML (possibly with some extensions or profiles) for defining interfaces. In the implementation stage, there are two common ways of defining interfaces.

One way is to describe interfaces by means of an interface description language (IDL) that is independent from a particular programming language. Through mappings between specific programming languages and the IDL, interoperability between multiple programming languages is achieved: components implemented in different programming languages can be combined into one system. IDLs focus only on syntactic interoperability, but they (implicitly, and sometimes unintentionally) also determine the styles of interaction through which components can communicate. The syntactic interoperability achieved by IDLs yields the benefit of using different programming languages for the component implementations.

Another way of specifying an interface is to directly use a programming language, as for example using an object-oriented language. Typically, in object-oriented programming languages, a component is expressed as a class in which the interface is defined as a set of methods and attributes, possibly with some extensions or syntactic convention to distinguish component architectural

elements (for instance required and provided interface). In other languages, the structured (stereotyped) use of header files or abstract classes serves as a means of defining interfaces.

Driven by the requirements of independent deployment and dynamic reconfiguration, some component models define a standard for the binary representation of interfaces. This binary representation is used at the deployment stage and during run-time. MS COM is an example of a component model that has such a binary standard for interfaces.

To make it possible to perform advanced checks on the compatibility between interfaces, the notion of contract has been adjoined to interfaces. According to [41], contracts can be classified hierarchically in four levels which, if taken together, may form a global contract. In our classification, we adopt the first three levels, since the last level is concerned with extra-functional properties which are covered in more detail in Section 2.4.

- *Syntactic level*: describes the syntactic aspect, also called signature of an interface. This level ensures that the interacting components refer to the same data types. This is the most common and most easy agreement to certify as it relies mainly on a (either static or dynamic) type-checking technique.

- *Functional Semantic level*: reinforces the previous level of contracts in certifying that the values of the parameters are within the proper range. This can be asserted using pre-conditions, post-conditions and invariants.

- *Behaviour level*: expresses either constraints on the temporal ordering of interactions between components or constraints on the component's internal behaviour (e.g. allowed internal states) in response to interactions. Behaviour contracts are typically expressed by statecharts or different variants of finite state machines.

We conclude our discussion on aspects of interfaces by pointing out that several component models have distinctive features related to evolvability and variability. For instance, for evolvability (e.g. to support creating new functionality but maintaining backward compatibility), a component may offer multiple interfaces for the same functionality. This makes it possible to embody several versions or variants of functions in the component.

Binding mechanisms

Binding is the process that establishes connections between components (through use of their interfaces and interaction channels). In CBSE, binding is also often called *component composition* by reference to the composition of the functionality of the components. Similarly by association to wires in electrical engineering, binding is also referred to as *wiring* in the literature e.g. [42] and [7].

An important question coming from the possibilities offered by binding mechanisms relates to the composability of components [28]: "Can an assembly, i.e. a set of components mutually connected, be treated as a component itself?". That is, does an assembly composed from a set of components fully comply to the rules imposed by the component model, both in terms of functional and extra-functional properties? The answer is not simple. To discuss component composition, we must first distinguish different types of binding: *horizontal binding* and *vertical binding* as defined below.

Let us assume that the following components $C_i = \langle I_i, P_i \rangle$ and $C_j = \langle I_j, P_j \rangle$ satisfy the rules imposed by a component model CM, i.e.

$$C_i, C_j \models CM \Rightarrow I_i, I_j, P_i, P_j \models CM$$

If we compose C_i and C_j together through an *horizontal binding* meaning that their respective interfaces are connected together (i.e. $\langle I_i \oplus I_j \rangle$), then the assembly A resulting of this composition is merely a set of components cooperating together to realize a functionality, i.e. $A = \{C_i, C_j\}$. Here, A does not necessary comply with the component model CM. In spite of this, this type of binding is often improperly referred to as horizontal composition. At the modelling stage, horizontal binding is often realized by connecting a provided interface of a component with a required interface of another component. At the implementation stage, this horizontal binding is typically realized through glue-code or wrappers.

On the other hand, if we identify the assembly A as a component with an interface I_A which is a composition of interfaces of the involved components, i.e. if we have

$$\begin{aligned} A = \{C_i, C_j\}; A = \langle I_A \rangle \Rightarrow I_A &= \langle I_i \oplus I_j \rangle \\ \text{where } I_A &\models CM \end{aligned}$$

then A results from a *vertical binding* and has an interface I_A that satisfies the rules of the component model CM. At the modelling stage, vertical binding is

often attained through connecting two interfaces of the same kind: a provided interface of the assembly (resp. required interface) to a provided interface of an inner component (resp. required interface). This type of connection is called *delegation*. Whereas when all the interfaces of the inner components are made available to the outside environment through the interfaces of the assembly, we speak of *aggregation*.

If the assembly A satisfies the component model's rules with respect to both its interface I_A and its properties P_A, i.e.

$$\begin{aligned} A = \langle I_A, P_A \rangle \quad &\Rightarrow \quad A = \langle I_i \oplus I_j, P_{ex} \vdash P_i \oplus P_j \rangle \\ \text{where} \quad I_A, P_A \quad &\models CM \end{aligned}$$

then the component model supports *vertical composition*. This is a very powerful property, but unfortunately very difficult to achieve in practice. Nevertheless, many component models support *partial vertical composition*, in which functional interfaces can be composed recursively.

In SaveCCM [14], vertical binding is supported and the component model defines an assembly as a set of components which export by delegation a set of selected ports, the interface elements. If the assembly also preserves the "read-execute-write" semantics defined by SaveCCM for components, then in that particular case, the assembly is a component because it complies with the definition of a SaveCCM component.

Binding does not necessarily correspond only to a one-to-one direct connection between two components; some component models also support indirect connections through the utilisation of connectors. When introduced as first class citizens of a component model, connectors act as mediators between components and enable (i) making the interaction between components explicit, and (ii) the addition (and removal) of advanced mediation mechanisms that are transparent to components. In several component models, connectors are implemented as special types of components (e.g. adaptors, brokers or proxies). Implementing connectors in terms of implementation-level components opens up the possibility of building more complex interactions patterns in comparison to using basic connectors.

The use of connectors corresponds to the concept of *exogenous composition* because the (logic for handling the) interaction between components is handled outside of the components themselves. In contrast to exogenous composition, *endogenous composition* refers to a binding without any intermediary connector. In this case, the handling of binding and interaction protocols is part of the components themselves.

At the modelling and implementation stages, binding is done by a system developer who explicitly states which components are assembled together by connecting the interfaces of the involved components. This is one of the forms of *third-party binding* in which the establishment of the binding is initiated by an entity outside the components involved in the binding. On the other hand, in a *first-party binding*, a component decides itself which other component it is to be bound to. Most of the component models enables the third-party binding. Typical solutions for first-party binding use an introspection (or reflexion) interface, which enables the discovery of the interfaces of the components to connect to, and a registry, which can look up the identity of the components that support a specific functionality (or interface).

When the binding occurs at deployment stage, a docking interface is commonly used. This docking interface does not offer any application functionality, but serves instead for managing the binding and subsequent interaction between a component and the underlying run-time infrastructure. In many component models (e.g. CCM, EJB), the binding specification is location-transparent: the run-time location of components (placed either on a local or a remote node) is specified separately from the binding information.

Interactions

Component models use one or more architectural styles following a specific *interaction styles* to define the patterns of interactions between components, i.e. how components communicate with each other. For instance, the client-server architectural style, widely used for distributed computing, uses a *request-response* interaction model. This means that for any interaction between two components, one component sends a request to a specific other component, which then returns a reply. Hence traffic across the binding is bidirectional.

Two variants of request-response are distinguished. In *asynchronous* request-response, the client initiates the communication, and continues its activity until, at some point, it receives the results of its request from the server component. The interaction can also be *synchronous*, which means that the client waits until its request has been processed.

Another typical interaction style is *pipe & filter*, which is mostly used for the streaming of events. This style uses unidirectional communication between components. In this style, components are filters that process the data, and the bindings are the pipes that transfer the data to the next filters. A characteristic of this style is that it allows the separate control of the data-flow and control-flow between components. The control flow is activated by a triggering interaction model, which enables the activation of a particular component in response to a particular signal such as an event, a clock tick, or a stimulus from another component, as illustrated in Fig. 2.3.b. This interaction model includes event-triggering, or event-driven, and time-triggering. The pipe & filter architectural style is widely used in embedded and real-time systems because control theory can be easily mapped to this interaction model. Some component models such as Rubus [11] decouple the specification of data flow from control flow.

There are other interaction styles utilized in component models, and some prominent examples are broadcast, blackboard and publish-subscribe. In most cases, component models provide a single basic interaction style. Support for this style is often hardwired in the execution platform. However, some component models, such as Fractal [43], Pin [32] and BIP [44] allow the construction of different interaction styles.

An interaction style determines which types of dependencies must or may exist between components. As a result, the architectural styles supported by a component model have a large impact on the flexibility during both the development and the execution of components. In general, a style which induces more or stronger dependencies will need more complex protocols for binding and hence for the replacement of components.

Components may differ with respect to the way their internal activity and interactions are initiated. *Passive components* are activated only by external events (for example being called by another component), whereas *active components* manage their activation themselves, and can be executed in a separate thread. Some component models provide support only for passive components (e.g. AUTOSAR, SaveCCM) while others have developed different ways for component startup and execution (e.g. CCM, MS COM). Often, the mechanisms for the activation of components are governed by the underlying middleware [45] or operating system, or are taken from the supporting implementation language.

Construction classification

In accordance with the observations and reasoning from above, we identify the following classification characteristics for interfaces and connections in the construction dimension.

C.1 *Interface specification*, in which different characteristics allowing the specification of interfaces are identified:

(a) The distinction of interface type: operation-based (e.g. methods invocations) and port-based interface (e.g. data passing).
(b) The distinction between the provides-part and the requires-part of an interface.
(c) The existence of some distinctive features.
(d) The language used to specify the interface.
(e) Interface levels which describe the levels of contractualisation of the interfaces, namely syntactic, functional semantic and/or behaviour level.

C.2 *Binding*, which describes the characteristics of the patterns and mechanisms used for binding components. It consists of two subtypes:

(a) The exogenous sub-category describes whether the component model includes connectors as architectural elements or not.
(b) The hierarchical sub-category expresses the possibility of having a hierarchical composition of components (horizontal composition is an intrinsic part of all component models, thus it is implicitly assumed to be supported).

C.3 *Interactions*, which comprise the following characteristics:

(a) Interaction style, which describes the main underlying architectural style used.
(b) Communication type, which details if the communication used is synchronous and/or asynchronous.

2.2.3 Extra-Functional Properties

Components and component-based systems are carriers of a number of extra-functional properties. The most basic support that a component model can provide for extra-functional properties is to facilitate specifying such extra-functional properties. For example in Robocop [15], components may specify the maximum execution time per method of an interface. A specification of such properties makes it possible to check at the component's deployment whether a component breaks the system integrity or requires more resources than the system can ensure.

Another type of support that a component model can provide is related to the management of particular extra-functional properties. For example, CCM [34] explicitly provides redundancy mechanisms for managing reliability.

Yet another type of support provided by component models is related to property compositions; it enables the prediction of systems properties derived from the properties of the integrated components and the underlying component framework.

In this section we discuss the EFP specification, management mechanisms and EFP composition issues, and then we identify the elements in the classification framework that make it possible to distinguish different component models.

Specification of extra-functional properties

Component models rarely address the specification of extra-functional properties (which by definition belongs to metadata). In many cases, extra-functional properties are specified implicitly, not as a part of a component model, but as a part of the component technology. A basic form of EFP specification is the one proposed by Mary Shaw [46], where an EFP is specified as a triple $\langle Attribute,\ Value,\ Credibility \rangle$ where *Attribute* describes the property itself, *Value* the corresponding data, and *Credibility* specifies the confidence in the value. The attribute *Value* is often a simple data type, but some component models provide a more complex value type (such as a reliability distribution). The Pin component model has an associated "Predictability-Enabled Component Technology (PECT)" [32] [47], which enables the specification and handling of the extra-functional properties through "analytical interfaces". Pin requires that a reasoning framework is specified which defines how to analyse a particular type of property. In Robocop [15], a resource model describes

the resource consumption of components in terms of mathematical cost functions, and a behavioural model specifies the sequence in which their operations must be invoked. Based on this information, associated analysis techniques can then analyse the total resource usage and response times. Similarly, Palladio [48] extends behaviour specifications with annotations (or extensions) of their resource usage, and their failure probabilities. Together with a model of the physical resources, performance and reliability metrics can be derived. Most of the component models define extra-functional properties as attributes of components or, more seldomly, as attributes of assemblies or of a systems.

Management of extra-functional properties

Component models provide different types of support for managing EFP. This management is related to run-time extra-functional properties and realised in combination of components and underlying component execution platform that can often be integrated as a part of a middleware. Different mechanisms for management of extra-functional properties (as well as for component deployments and communication mechanisms) can be found in [45]. We distinguish four types of support (see Fig. 2.4):

1. *Exogenous Management*. The EFP management is provided outside the components.

2. *Endogenous Management*. The EFP management is implemented in the components, i.e. the component developers are responsible to implement it.

3. *Management per Collaboration*. The EFP management is realized in direct interactions between components.

4. *Systemwide Management*. The EFP management is provided by the component framework, or underlying middleware.

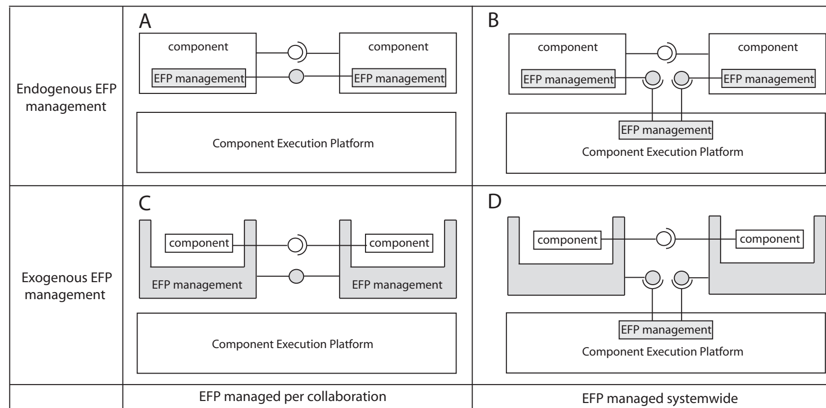


Figure 2.4: Management of extra-functional properties

By a combination of these types we get four possible types of the EFP support:

- *Approach A* (*endogenous per collaboration*). A component model does not provide any support for EFP management, but it is expected that a component developer implements it. This approach makes it possible to include EFP management policies that are optimized towards a specific system, and also can cater for adopting multiple policies in one system. This heterogeneity may be particularly useful when COTS components need to be integrated. On the other hand, the fact that such policies are not standardized may be a source of architectural mismatch between components. A risk of using this approach is a hetereogeneity of policies for handling a single EFP in a system. As a result, managing and predicting emerging properties at the system level can be very difficult.

- *Approach B* (*endogenous systemwide*). In this approach, there is a mechanism in the component execution platform that contains policies for managing extra-functional properties for individual components as well as for extra-functional properties involving multiple components. The ability to negotiate the manner in which extra-functional properties are handled requires that the components themselves have some know-ledge about how the extra-functional properties affect their functioning. This is a form of reflection applied to EFP management.

- *Approach C* (*exogenous per collaboration*). In this approach, components are designed such that they address only functional aspects and are oblivious to EFP. Consequently, in the execution environment, these components are surrounded by a container. This container contains the knowledge on how to manage extra-functional properties. In this approach, containers are connected to other containers. Connected containers then manage the extra-functional properties for the components that they encapsulate.

  The container approach is a way of realizing the separation of concerns in which components concentrate on functional aspects and containers concentrate on extra-functional aspects. In this way, components become more generic because no modification is required to integrate them into systems that may employ different policies for extra-functional properties. Because these components do not address extra-functional properties, they are simpler to implement. A disadvantage of the container approaches might be a degradation of the system performance.

- *Approach D* (*exogenous system-wide*). This approach is similar to approach C, except that the system can coordinate the management of an EFP from a global system-wide perspective (e.g. global load balancing). Consequently, a more complex support need to be built into the component execution platform.

Composition of extra-functional properties

The most difficult challenge in CBSE is related to composing extra-functional properties. Compositions of extra-functional properties are based on different composition theories, and, in addition, they are often not only the result of compositions of component properties, but also depend on other elements of a particular system architecture or even its environment. For example, determining the composition of component performance may depend on the scheduling policies and the system architecture. According to [28], extra-functional properties can be classified in categories depending on the composition domains (i.e. type of parameters that determine the composition). The following categories are proposed:

- *Directly composable properties*: A property p_k of an assembly $A = \langle C_1 \oplus C_2 \rangle$ is a function of, and only of, the same property of the components involved.

  $$p_k(A) = f(p_k(C_1), p_k(C_2))$$

  An example of such property is static memory consumption. In the simplest case, the system static memory is the sum of component static memories plus a constant.

- *Architecture-related properties*: A property p_k of an assembly $A = \langle C_1 \oplus C_2 \rangle$ is a function of the same property of the components and of the software architecture SA.

  $$p_k(A) = f(SA, p_k(C_1), p_k(C_2))$$

  An example of such property is performance: increasing the amount of parallel processing impacts the performance of the system without changing the properties of individual components (for details see [28]).

- *Emerging properties*: A property p_k of an assembly $A = \langle C_1 \oplus C_2 \rangle$ depends on several different properties p_i, p_j of the components and of the software architecture.

  $$p_k(A) = f(SA, p_i(C_1), p_i(C_2), p_j(C_1), p_j(C_2) \dots)$$

  An example of an emerging property is response time of an assembly which depends on the execution time and resource consumption of the involved components.

- *Usage-depended properties*: A property of an assembly is determined by its usage profile U.

  $$p_k(A, U) = f(SA, \dots p_i(C_j, U_j) \dots)$$

  Reliability is an example of such property type. The reliability of a same system can be different for the different usage profiles of that system.

- *System environment context properties*: A property of a system S is determined by other properties and by the state of the system context X defined by external parameters outside the system.

  $$p_k(S, U, X) = f(SA, X, \dots p_i(C_j, U_j) \dots)$$

  Examples of this type are security and safety. These properties depend also on external conditions (such as different measures and procedures).

- *Non-composable properties*: Properties that are not composable. Examples of such properties are maintainability, robustness, portability, etc.

This classification indicates the limitations of the compositions of extra-functional properties. In general, determining the compositions of component properties becomes feasible only when restrictions are imposed on the design of individual components. In practice, such restrictions are imposed by the rules/constraints of the component model and system architecture. For example, static memory usage of an assembly can be defined as the sum of static memory usage of involved components, but only using particular composition policies (e.g. no concurrency). Other properties are related to usage profile, and if we cannot predict/specify the usage profile, we cannot predict the system properties.

Extra-functional properties classification

For the extra-functional properties, we provide a classification with respect to the following questions:

E.1 *Management of EFPs*: Which type of management (if any) is provided by the component model?

E.2 *EFP specification*: Does the component model contain means for the specification of specific EFPs? If yes, which properties and in which form?

E.3 *Composability of EFPs*: Does the component model provide means, methods and/or techniques for the composition of certain extra-functional properties and/or what type of composition?

2.2.4 The Classification Overview

Fig. 2.5 summarizes the classification framework in a graph form. The numbered items that describe the classification elements of the three dimensions are listed in the figure.

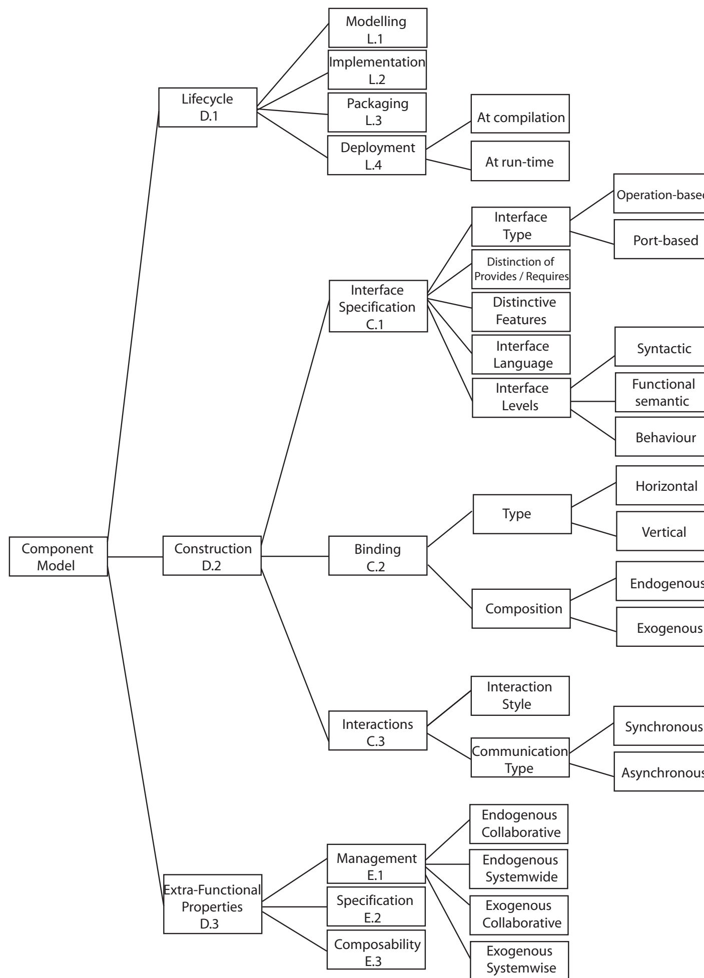


Figure 2.5: The hierarchical structure of the classification framework

2.3 Surveying Existing Component Models

Using the classification framework, we can analyze component models developed in different research groups or in industry. In our classification of component models, the first question is whether a particular approach (model, technology or method) is a component model or not. This appeared to be a difficult task due to the diversity of component models. Similar to biology, where viruses straddle the border between life and non-life, there is a wide range of models, from those having many elements of component models, yet not being considered component models, via those that lack many elements, but still are designated as component models, to those that are widely accepted as component models. Therefore, we identify the minimum criteria required to classify an approach as a component model.

The minimum criteria correspond to the definition of component models given in the introduction and in Section 2.1:

1) A component model includes a component definition;
2) A component model provides rules for component interoperability;
3) Component functional properties are unambiguously specified by component interface;
4) A component interface is used in the interoperability mechanisms;
5) A component is an executable piece of software and the component model either directly specifies its form or unambiguously relates to it via interface and interoperability specification.

Note that the items from the “lifecycle” and “construction” dimensions from the classification framework belong in the minimum criteria, while extra-functional properties are not included in the minimum, and many component models do not provide that support.

There is a wide range of approaches that comply with some of the elements in the minimum criteria. For example, many modelling languages have “components” and even (semi-)formally specify components and component compositions. For instance in ADLs, the basic elements are components [49]. UML 2.0 provides a metamodel for components, interfaces and ports. Still, we have deliberately chosen not to select them as component models, in contrast to other classifications such as [50]. One reason is that their purpose is not

component-based development, but rather the specification of system architectures, and they do not provide any support for components as executable units. Certain languages derived from UML, such as xUML [51], in which the component specification is translated into an executable entity, are even stronger candidates for consideration as component models. However, xUML and similar languages do not operate with components as first class entities (for example components are not treated as separate development or executable entities), but again the components are treated only as architectural elements.

On the other side of the lifecycle line are services. It can be argued that services are special types of components. Services are focused on run-time retrieval and run-time deployment. Similar to components, services are specified by an interface, and provide support for construction [52]. Still, we have not included services in the classification for similar reasons as those that applied to ADLs — they are not defined as executable units. In analogy to ADLs, services are not component models but rather use component models.

2.3.1 Component Model Selection

In our classification framework, we have selected 24 component models that we encountered in the research literature and in practice, namely:

- **AUTOSAR** (AUTomotive Open System ARchitecture) [8], a new standard architecture created by a partnership between several manufacturers and suppliers from the automotive field.
- **BIP** (Behaviour, Interaction, Priority) [44], a framework developed at Verimag for modelling heterogeneous real-time components.
- **BlueArX** [10], a component model developed and used by Bosch for the automotive control domain.
- **CCM** (CORBA Component Model)] [34], a part of the CORBA 3 standard defined by Object Management Group (OMG).
- **COMDES II** (COMponent-based design of software for Distributed Embedded Systems, version II) [53], a component-based software framework aimed for efficient development of reliable distributed embedded control systems with hard real-rime requirements.
- **CompoNETS** [54], a general-purpose component model developed at the Université de Toulouse 1 that uses high-level Petri-Nets for behaviour modelling.

- **EJB** (Entreprise JavaBeans) [33], a component model developed by Sun MicroSystems.
- **Fractal** [43], a component model developed by France Telecom R&D and INRIA.
- **Koala** [12], a component model developed by Philips for building software for consumer electronics.
- **KobrA** (KOmponentenBasieRte Anwendungsentwicklung)] [31], a general-purpose software engineering method for the development of component-based application frameworks.
- **IEC 61131** [40], a standard for the design of Programmable Logic Controllers approved by the International Electrotechnical Commission.
- **IEC 61499** [55], a standard developed by the International Electrotechnical Commission to support the development of automation and control systems.
- **JB** (Java Beans) [37], a portable, platform-independent software component model for the Java Standard Edition platform.
- **MS COM** (Microsoft Component Object Model) [35], one of the most commonly used general-purpose component model for desktop and server side applications.
- **OpenCOM** [56], a lightweight component model developed at Lancaster University.
- **OSGi** (Open Services Gateway Initiative) [38], a consortium of industrial partners working together to define a service-oriented framework with open specifications.
- **Palladio** [48], a component model developed at Karlsruhe Institute of Technology and FZI Karlsruhe for early performance predictions of component-based software architectures of business information systems.
- **Pecos** (PErvasive COmponent Systems) [57], a joined project between ABB Corporate Research and Bern University that provides a component model for the development of software for field devices.

- **Pin** [32], a component model, developed at Carnegie Mellon Software Engineering Institute (SEI),to serve as basis in prediction-enabled component technologies (PECTs).

- **ProCom** (PROGRESS Component Model) [22], a component model for control-intensive distributed embedded systems developed at Mälardalen University.

- **Robocop** (Robust Open Component Based Software Architecture for Configurable Devices Project) [15] [58], a component model developed by the consortium of the Robocop ITEA project, inspired by COM, CORBA and Koala component models.

- **Rubus** [11], a component model developed as a joint project between Arcticus Systems AB and Mälardalen University for development of distributed, resource-constrained, embedded control systems, with a mix of hard-, soft- and non real-time requirements.

- **SaveCCM** (SAVE Components Component Model) [14], a component model for predictable embedded control applications in the automotive domain, developed as a collaboration between several Swedish universities.

- **SOFA** (Software Appliances) [59], a component model developed at Charles University in Prague.

While some of these component models are in widespread industrial use, others are used as demonstrators or vehicles for illustrating research ideas. The classification framework does not show the success of particular component models, or any business model, but is based only on their technical characteristics. The component models that we have included in the list are briefly characterized [17]. A more detailed description of each component model with the characteristics defined in the classification framework can be found in a technical report in [60].

For some of the component models that we found, our selection criteria were satisfied; however, because of the scarcity of available documentation about some component models, it was impossible to get the necessary detailed information (which usually is a sign that no activity around the model is going on). In these cases, we have decided to omit them from our list.

2.3.2 Methodology

Our research methodology followed an empirical approach consisting of the successive iterations of the steps of: (i) observations and analysis, (ii) classification, and (iii) validation. The observations and analysis included studying of a number of component models and the literature related to the general principles of CBSE [61, 42, 62, 27, 29, 7, 41], and related classifications [49, 28, 63, 50]. In addition, we utilized our own experience gained from the development of the SaveCCM [14], ProCom [22], and Robocop [15] component models, and our tight cooperation with industry that used some component technologies in their development (ABB (COM, Pin), Ericsson (Service-oriented architecture), Philips (Koala, Robocop), Volvo (AUTOSAR, Rubus), Arcticus (Rubus)). Based on this, our classification framework was built, incrementally populated and refined with a set of component models. The validation consisted of trying to fit at each iteration a larger set of component models into the framework. Further validation was performed by discussing the framework with several CBSE-experts from industry and academia and with researchers in the broader field of software engineering. For several component models, we contacted their developers and obtained feedbacks on the classification we proposed for "their" component models. The resulting analysis and discussions have also led to a refinement of the framework.

2.4 The Comparison Framework

The characteristics of the component models are collected in the tables below, following the dimensions in the classification framework, namely lifecycle (Table 2.1), construction (Tables 2.2 and 2.3), and extra-functional properties (Table 2.4). Following each table, a short discussion summing up our observations is presented.

2.4.1 Lifecycle Classification

Table 2.1 shows the lifecycle dimension, indicating the characteristics of the selected component models in different lifecycle stages (modelling, implementation, packaging and deployment).

Table 2.1: Classification for the Lifecycle Dimension

Component Models	Modelling	Implementation	Packaging	Deployment
AUTOSAR	N/A	C	Non-formal specification of container	At compilation
BIP	A 3-layered representation: behaviour, interaction, and priority	BIP Language	N/A	At compilation
BlueArX	ASCET-MD models	C	Packages	At compilation
CCM	N/A	Language independent	JARs, DLLs	At run-time
COMDES II	ADL-like language	C	N/A	At compilation
CompoNETS	Petri Nets	Language independent	JARs, DLLs	At run-time
EJB	N/A	Java	JARs	At run-time
Fractal	ADL-like language (Fractal ADL, Fractal IDL), Annotations (Fractlet)	Java (Julia, Aokell) C/C++ (Think) .Net lang. (FracNet)	File system based repository	At run-time
Koala	ADL-like languages (IDL,CDL and DDL)	C	File system based repository	At compilation
KobrA	UML Profile	Language independent	N/A	N/A
IEC 61131	Function Block Diagram (FBD) Ladder Diagram (LD) Sequential Function Chart (SFC)	Structured Text (ST) Instruction List (IL)	N/A	At compilation
IEC 61499	Function Block Diagram (FBD)	Language independent	N/A	At compilation
JavaBeans	N/A	Java	JARs	At compilation
MS COM	N/A	OO languages	DLLs	At compilation and at run-time
OpenCOM	N/A	OO languages	DLLs	At run-time
OSGi	N/A	Java	JARs	At compilation and at run-time
Palladio	Meta-model based specification language	Language independent (specific support for Java)	N/A	N/A

Table 2.1: Classification for the Lifecycle Dimension

Component Models	Modelling	Implementation	Packaging	Deployment
PECOS	ADL-like language (CoCo)	OO languages	JARs, DLLs	At compilation
Pin	ADL-like language (CCL)	C	DLLs	At compilation
ProCom	Meta-model based specification language REMES	C	File system based repository	At compilation
ROBOCOP	Meta-model based specification language Resource management model	C and C++	ZIP file	At compilation and at run-time
RUBUS	Rubus Design Language	C	File system based repository	At compilation
SaveCCM	ADL-like (SaveComp) Timed automata	C, Java	File system based repository	At compilation
SOFA 2.0	Meta-model based specification language	Java	File system based repository	At run-time

From this table, we can observe that the most common focus of component models is on the implementation stage. Some component models even exclusively support the implementation stage. Additionally, some component models support the run-time stage by providing a run-time platform that facilitates run-time reconfiguration or a management of extra-functional system properties.

The modelling stage is characterized by an extensive use of domain-specific modelling languages, whereas standard modelling language, such as UML or ADLs are less common. We can also note that 32% of the component models gathered in the framework do not provide any support for the modelling of components or component-based applications, but cover only the implementation part (specification and deployment). All these component models that omit the modelling stage are from the state of the practice, and many of them widely used. One can ask why component models in practice seldom cover component and system modelling. The reason for this can be found in the common state-of-the-practice. In many industrial projects, designs are expressed in a non-formal way, mainly for documentation purpose only, or in a semiformal way, possibly using UML. In both cases, neither the precise definitions of components nor their interactions are assumed to be of high priority,

and no high needs for modelling components and component-based systems are expressed. This is also an indicator of the differences between state-of-the-art and state-of-the-practice: many solutions from the state-of-the-art that include the modelling have still not been realized or scaled up in practice.

Further, we can observe from Table 2.1 that with regards to implementation, component models can be divided into four groups: *i*) language-independent (18%), *ii*) OO language-based (36%), with a clear dominance of Java, *iii*) C language (36%), and *iv*) domain-specific language-based (10%), either compiled to C or directly interpreted. The dominance of OO languages is not surprising since technologies based on the OO paradigm are dominant today, and because many principles from OO are directly used or further developed in CBSE. The "C language" component models are prevailing for domain-specific component models that target more the development of embedded and real-time systems. The C-language provides more and easier access to details of operating system and underlying hardware platforms facilitating optimisations. Domain-specific programming languages are tightly related to the modelling of component-based systems and components, and obviously used for a more efficient design and implementation.

Packaging and component repositories are not the main focus of component models. In most cases, certain standard archives are used (such as DLL or JAR packages), also as deployment units. The lack of repositories indicates a low focus on reuse, in particular of COTS components.

Deployment at compile time and run-time almost occurs to an equal extent among the component models being studied. Deployment at compile time limits the flexibility at run-time, but on the other hand enables easier predictability, richer composition features (such as hierarchical composition), and more efficient reuse (such as deployment of implementation parts that will be used in the application). This might be a reason why this is the primary deployment style chosen by specialized component models (see Table 2.5). From this table, we can observe that the most common focus of component models is on the implementation stage. Some component models even exclusively support the implementation stage. Additionally, some component models support the run-time stage by providing a run-time platform that facilitates run-time reconfiguration or a management of extra-functional system properties.

2.4.2 Construction Classification

Table 2.2 presents the interface characteristics of the selected component models, and Table 2.3 the binding and interaction specifications. Table 2.2 shows that most of the interfaces are of the operation-based type, which means that the component models use methods and parameters for defining interface signatures. Still, many component models use ports as the interface elements to exchange data. In port-based interfaces, input and output interfaces consist of ports that receive and send data, respectively (often designated as sink and source), hence corresponding to the concepts of provided and required interface. Such component models are typically used in embedded systems and have their basis in hardware components. Several of the component models examined do not distinguish required from provided interfaces, but their interface is referred only to the "provided" interface, which is similar to what exists in the object-oriented approach. These component models are essentially used in practice, and are developed earlier, even on the way to becoming obsolete (like MS COM, for example). They illustrate the evolution of CBSE.

Because interfaces are a mandatory part of the component specification, all component models provide at least the first level, i.e. syntactic specification. A considerable number of component models also have behaviour specifications, in most cases represented by a particular form of finite state machines (statecharts or timed automata). Here we distinguish behaviour specification of components (used for the modelling and predictability of the behaviour of the system), from specifications used for synchronization (for the communication between the components). In a few cases, component models allow behaviour specification with resource consumption to be combined, or some other attribute specifications, which makes it possible to model resource usage or performance or some other properties. Examples of such component models are Palladio, SaveCCM, ProCom, and Pin. Only few component models offer support for defining the functional semantic level of interfaces. If there is support, then this is mostly addressed through the use of pre- and post-conditions.

Table 2.2: Classification for the Construction Dimension
Interface Specification

Component Models	Interface type	Distinction of Provides/ Requires	Distinctive features	Interface Language	Interface Levels (Syntactic, Semantic, Behaviour)
AUTOSAR	Operation-based Port-based	Yes	AUTOSAR interface	C header files	Syntactic
BIP	Operation-based Port-based	No	Complete interface Incomplete interface	BIP Language	Syntactic Semantic Behaviour
BlueArX	Port-based	Yes	Configuration interface Analytic interface	XML adhering to the MSRSW DTD	Syntactic
CCM	Operation-based Port-based	Yes	Facet and receptacle Event sink and event source	CORBA IDL (CIDL)	Syntactic
COMDES II	Port-based	Yes	N/A	C header files State charts diagrams	Syntactic Behaviour
CompoNETS	Operation-based Port-based	Yes	Facet and receptacle Event sink and event source	CORBA IDL (CIDL) Petri nets	Syntactic Behaviour
EJB	Operation-based	No	N/A	Java Programming Language + Annotations	Syntactic
Fractal	Operation-based	Yes	Component interface Control interface	IDL, Fractal ADL, Java or C Behavioural Protocol	Syntactic Behaviour
Koala	Operation-based	Yes	Diversity interface Optional interface	IDL, CDL	Syntactic
KobrA	Operation-based	N/A	N/A	UML	Syntactic
IEC 61131	Port-based	Yes	N/A	N/A	Syntactic
IEC 61499	Port-based	Yes	Data Event	N/A	Syntactic
JavaBeans	Operation-based	Yes	N/A	Java	Syntactic
MS COM	Operation-based	No	Ability to extend interface	Microsoft IDL	Syntactic

Table 2.2: Classification for the Construction Dimension
Interface Specification

Component Models	Interface type	Distinction of Provides/ Requires	Distinctive features	Interface Language	Interface Levels (Syntactic, Semantic, Behaviour)
OpenCom	Operation-based	No	Interfaces additional to COM-interface managing lifecycle, introspections, etc.	Microsoft IDL	Syntactic
OSGI	Operation-based	Yes	Dynamic interface	Java	Syntactic
Palladio	Operation-based	Yes	Parametrization	Palladio language (similar to CORBA IDL)	Syntactic Behaviour
PECOS	Port-based	Yes	Ability to extend interface	Coco language Prolog query Petri nets	Syntactic Semantic Behaviour
Pin	Port-based	Yes	N/A	Component Composition Language (CCL), UML statechart	Syntactic Behaviour
ProCom	Port-based	Yes	Data and trigger port Message port	XML based, REMES	Syntactic Behaviour
Robocop	Port-based	Yes	Ability to extend and annotate interface	Robocop IDL (RIDL), Protocol specification	Syntactic Behaviour
RUBUS	Port-based	Yes	Data and trigger port	C header files	Syntactic
SaveCCM	Port-based	Yes	Data, trigger, and data-trigger port	SaveComp (XMLbased) Timed Automata	Syntactic Behaviour
Sofa 2.0	Operation-based	Yes	Utility interface Possibility to annotate interface and control evolution	Java SPC algebra	Syntactic Behaviour

Table 2.3 (binding and interactions) shows that binding mechanisms in component models are in most of the cases of the endogenous type — i.e. connectors are not defined as particular architectural elements. However, many component models use components as connectors or the connectors are automatically generated in the integration/deployment stage and are not being used as entities for modelling.

We can also observe that many component models do not support vertical binding. Vertical binding is implemented either through delegated interfaces (i.e. selected interfaces from sub-components build up the interface of the composite components) or as aggregation in which the composite component includes all the interfaces of the aggregated components. Very few component models provide means of hierarchical composition, and if so, then it is only with regards to few particular extra-functional properties (for example BIP and SaveCCM for timing properties).

From the information in Table 2.3, one can conclude that the dominating interaction styles in component models are "request-response" (typically used in client/server architectures), and "pipe & filter". Some component models even have additional interaction styles such as event-driven, broadcast or rendezvous. The choice of the interface style is strongly correlated to the interface type (operation vs. port-based) provided by the component model.

The dominant communication type in component models is synchronous. Component models that provide support for asynchronous communication also support synchronous communication. This indicates that component models are not concerned with architecture (architectural design), but rather with targeting detailed design.

Table 2.3: Classification for the Construction Dimension
Binding and Interactions

Component Models	Binding		Interactions	
	Exogenous	Vertical	Interaction Styles	Communication Type
AUTOSAR	No	Delegation	Request-Response, Sender-Receiver	Synchronous, Asynchronous
BIP	No	Delegation	Triggering, Rendez-vous, Broadcast	Synchronous, Asynchronous

Table 2.3: Classification for the Construction Dimension
Binding and Interactions

Component Models	Binding		Interactions	
	Exogenous	Vertical	Interaction Styles	Communication Type
BlueArX	No	Delegation	Sender-Receiver, Request-Response	Synchronous, Asynchronous
CCM	No	No	Request-Response, Triggering	Synchronous, Asynchronous
COMDES II	No	No	Pipe&filter	Synchronous
CompoNETS	No	No	Request-Response	Synchronous, Asynchronous
EJB	No	No	Request-Response	Synchronous, Asynchronous
Fractal	Yes	Delegation, Aggregation	Multiple interaction styles	Synchronous, Asynchronous
Koala	No	Delegation, Aggregation	Request-Response	Synchronous
KobrA	No	Delegation, Aggregation	Request-Response	Synchronous
IEC 61131	No	Delegation	Pipe&filter	Synchronous
IEC 61499	No	Delegation	Triggering, Pipe&filter	Synchronous
JavaBeans	No	No	Request-Response, Triggering	Synchronous
MS COM	No	Delegation, Aggregation	Request-Response	Synchronous
OpenCOM	No	Delegation, Aggregation	Request-Response	Synchronous
OSGi	No	No	Request-Response, Triggering	Synchronous
Palladio	Yes	Delegation	Request-Response	Synchronous, Asynchronous
PECOS	No	Delegation	Pipe&filter	Synchronous

Table 2.3: Classification for the Construction Dimension Binding and Interactions

Component Models	Binding		Interactions	
	Exogenous	Vertical	Interaction Styles	Communication Type
Pin	No	No	Request-Response, Message passing, Triggering	Synchronous, Asynchronous
ProCom	Yes	Delegation	Pipe&filter, Message passing	Synchronous, Asynchronous
Robocop	No	No	Request-Response	Synchronous, Asynchronous
Rubus	No	No	Pipe&filter	Synchronous
SaveCCM	No	Delegation, Aggregation	Pipe&filter	Synchronous
SOFA 2.0	Yes	Delegation	Multiple interaction styles	Synchronous, Asynchronous

2.4.3 Extra-Functional Properties Classification

Table 2.4 summarizes the characteristics of the selected component models with respect to extra-functional properties. We observe that many component models provide certain support for the management of extra-functional properties, either system-wide or per container (characteristic examples are redundancy, or authentication support). In several cases, a particular EFP support is implemented as an extension to a standard technology (for example COM+ used in MS COM and .NET technologies). However, a smaller number of component models have formalisms for EFP specifications. A significantly smaller number of component models provides means for the composition of extra-functional properties. This is particularly true for commercial component models. Clearly, the composition of extra-functional properties still belongs to the research challenges. A majority of extra-functional properties that are managed by component models belong to resource usage and timing properties.

Table 2.4: Classification for the Extra-Functional Properties Dimension

Component Models	Management of EFP	EFP specification	Composability of EFP
AUTOSAR	Endogenous per collaboration (A)	N/A	N/A
BIP	Endogenous system wide (B)	Timing properties	Behaviour compositions
BlueArX	Endogenous system wide (B)	Resource usage and timing properties	Reasoning frameworks
CCM	Exogenous system wide (D)	N/A	N/A
COMDES II	Endogenous system wide (B)	Timing properties	N/A
CompoNETS	Endogenous per collaboration (A)	N/A	N/A
EJB	Exogenous system wide (D)	N/A	N/A
Fractal	Exogenous per collaboration (C)	Ability to add property (by adding property controller)	N/A
Koala	Endogenous system wide (B)	Resource usage	Compile time checks of resources
KobrA	Endogenous per collaboration (A)	N/A	N/A
IEC 61131	Endogenous per collaboration (A)	N/A	N/A
IEC 61499	Endogenous per collaboration (A)	N/A	N/A
JavaBeans	Endogenous per collaboration (A)	N/A	N/A
MS COM	Endogenous per collaboration (A)	N/A	N/A
OpenCOM	Endogenous per collaboration (A)	N/A	N/A
OSGi	Endogenous per collaboration (A)	N/A	N/A
Palladio	Endogenous system wide (B)	Performance, reliability, resource usage, system-level usage properties	Performance and reliability
PECOS	Endogenous system wide (B)	Generic specification of properties including timing properties	N/A
Pin	Exogenous system wide (D)	Timing properties (by adding analytic interface)	Different EFP composition theories (ex: latency)

Table 2.4: Classification for the Extra-Functional Properties Dimension

Component Models	Management of EFP	EFP specification	Composability of EFP
ProCom	Endogenous system wide (B)	Generic specification of properties including timing and resource usage	Timing and resource usage properties at design and compile time
ROBOCOP	Endogenous system wide (B)	Memory consumption, timing properties, reliability Ability to add other properties	Memory consumption and timing properties at deployment
RUBUS	Endogenous system wide (B)	Timing properties	Timing properties at design time
SaveCCM	Endogenous system wide (B)	Generic specification of properties including timing properties	Timing properties at design time
SOFA 2.0	Endogenous system wide (B)	Behavioural (protocols)	Composition at design

2.4.4 Component Models and Domains

The characteristics listed in the classification framework show some patterns: similar solutions belong to component models from similar application domains, as for instance embedded systems or information systems. That is to say that the requirements from the application domain penetrate into the component model. Such component models are, as a consequence, specialised and not so usable in domains that are subject to different requirements.

The other type of component models that have similar solution patterns are general-purpose component models. They provide basic mechanisms for the specification and composition of components, but do not assume any specific architecture beyond general assumptions (like interaction style, support for distributed systems, compilation or run-time deployment). A general solution that enables component models to be both generally applicable and to cater for specific domains is the use of optional frameworks.

According to this, we distinguish the component models as:

- general-purpose component models;
- specialized component models.

Table 2.5: General-purpose and domain-specific component models

Domain	AUTOSAR	BIP	BlueArX	CCM	COMDES II	CompoNETS	EJB 3.0	Fractal	Koala	KobrA	IEC 61131	IEC 61499	JavaBeans	MS COM	OpenCOM	OSGi	Palladio	PECOS	Pin	ProCom	Robocop	Rubus	SaveCCM	SOFA 2.0
General-purpose				X		X	X	X		X			X	X	X		X		X					X
Specialised	X	X	X		X				X		X	X				X		X		X	X	X	X	

Table 2.5 lists the selected component models according to their dominant use in particular domains.

We see that the distribution between general-purpose component models and specialized component models is equal. It is likely that there are more specialized, proprietary component models that are not published. We have also observed a migration of certain component models. For example, OSGi was originally designed for embedded systems, but later has been used as general-purpose component model in different domains. Conversely, general-purpose component models have been adapted for particular domains by the addition of new features or by applying some restriction to certain functions.

Specialized component models from our selection belong to two domains: a) embedded systems, and b) distributed information systems. Component models from the embedded systems domain have some common characteristics: the "pipe & filter" interaction style is used, components are usually deployable at compilation time, resource-aware, and often there is support for the management of timing properties. These component models are significantly different from general-purpose component models. The component models from the information systems domains are more similar to general-purpose component models. Typically, they have similar characteristics as general-purpose component models, such as the use of "request-response" interaction style, support for run-time deployment, expandable interface, and implementation in object-oriented languages. Component models that target information systems differ from general-purpose component models through specific support for distributed components, data transaction support, interoperability with databases, and some architectural solutions such as redundancy or location transparency. In some cases, an extension of a component model is used for its specialization (for example, COM+ is an addition to COM used for distributed component-based systems).

Some general-purpose component models have a special feature; they have mechanisms for generating new component models. They provide a set of common principles and mechanisms to add new features, or change the existing ones (for example different implementation mechanisms for bindings or interactions). An example of these “generative” component models is Fractal. Fractal supports several variants of particular component model elements — for example, different type of binding and interaction, and the use of different programming languages (Fractal has Java-based and C-based implementations). Another example of such component model is Robocop. It provides a mechanism for adding different elements of the model (such as modelling languages, implementations, metadata in a form of documentation, and management for extra-functional properties). A particular instance of a Robocop is a component model that includes selected elements.

From the characteristics defined in the tables, we can observe that although there are many component models, they show similar patterns within the same or related domains. We can conclude that this gives us a good basis to converge different component models into a smaller number of component models dedicated to domain-specific requirements.

2.5 Conclusions

In this chapter, we have first presented a thorough study of the main concepts related to the notion of component models. Using this as a basis, we have derived a framework that allows classifying and comparing component models according to these concepts. The intention of this work is to increase the understanding of the component-based approach by identifying the main concerns, common characteristics and differences of component models. The proposed framework does not include all the elements of all component models since many of them have unique solutions. However, the framework identifies minimal criteria for considering a model to be a component model, groups the characteristics into dimensions and enables a more systematic approach for their analysis and comparison.

From the use of the classification framework with a set of twenty-four selected component models, the following conclusions that are of interest for the thesis can be drawn:

1) **All the principles promoted by CBD are not always included in all component models**. This means that, there is, currently, no complete set of principles that applies to all component models. Many of the principles used in component models are directly taken from other approaches, such as object-oriented development, and ADLs, and further developed. As a result, this provides diverse solutions for similar approaches.

2) **Common patterns exist between component models from the same domain**. For example, general-purpose component models utilize the "request response" style, while in the specialized domains (mostly embedded systems) "pipe & filter" is the predominant style. Similarly, the "C language" is prevailing for component models that specifically target the development of embedded and real-time system.

3) **A generic support for specifying and composing extra-functional properties is currently lacking**. Few component models have formalisms for EFP specifications, and significantly fewer for their compositions. There are several reasons for that: in practice, explicit modelling and reasoning about of extra-functional properties is still not widespread; furthermore, many different extra-functional properties exist and many of them are not composable, or not directly composable but instead depend on external factors such as underlying platform, usage scenario, or the context in which the system is running.

Chapter 3

Defining Multi-Valued Context-Aware Extra-Functional Properties

As identified in Chapter 2, a few component models provide support for specification and management of extra-functional properties throughout the development process. In most cases, this support is limited to a single phase and unlike the well-established solution of embodying functionalities into interfaces, no consensus has emerged on how to handle extra-functionality in component models. When this support exists, it takes different forms: additional interfaces, annotations, or a language separated from the component models. These challenges on extra-functional properties management can be explained by the many aspects of extra-functional properties and the specific requirements of their usage within component-based development which increases the complexity of their management even more. Accordingly, the purpose of this chapter is to:

- Identify the various aspects and the corresponding challenges that must be taken into consideration for expressing, assessing and using extra-functional properties in component-based development.
- Define the key concept of multi-valued context-aware extra-functional properties to enable their management in component models, and by extension, in component-based development.

3.1 Extra-Functional Properties in Component-Based Development

In this section, we examine various aspects of extra-functional properties relevant for component-based development, namely their heterogeneity, their multi-valued nature and their context-sensitivity. Reciprocally, we also correlate aspects of component-based development, that are reusability and the separation between component type and instances, with extra-functional properties. For each of these aspects, we identify several challenges that must be addressed to provide a suitable management support for extra-functional properties in component-based development.

3.1.1 An Heterogeneous Data Set

Extra-functional properties provide additional information about the components, complementing the structural information that is provided by the component model. This additional information is intended to give a better insight in the behaviour and capability of the component in terms of reliability, safety, security, maintainability, accuracy, compliance to a standard, resource consumption, and timing capabilities, among many others. As stated in [28], the exhaustive list of possible extra-functional properties to consider is endless and there is no a priori, logical or conceptual method to determine which properties exist in a system or in components. Due to this, there is currently no unique list of extra-functional properties.

This problem inheres in one of the fundamental characteristics of extra-functional properties and properties in general: they are issued by humans. Therefore, different users will consider different types of information important for the development of the software system, and for the same property they might associate a different meaning and representation. For example, the worst-case execution time (WCET) is commonly defined as *"the longest execution time of a program that could ever be observed when the program is run on its target hardware"* [64]. However, since it is not possible to obtain the actual WCET, estimations such as over-, under- and probabilistic approximations are often implicitly used as a substitute. Yet, these values are fundamentally different. This difference is important to know in hard real-time systems, for instance, since for this type of systems the timing properties must be ensured.

The same is true about the representation of their value (data representation). A WCET can be expressed in standard time units such as milliseconds, or clock cycle. On the other hand, a parametric WCET is expressed in terms of formula which parameters are criteria that influence the value.

Further, different techniques can be used to assess extra-functional property value and different techniques often produce different results. Continuing with the WCET example, the WCET value can be assessed through different techniques as surveyed in [65]: static analysis methods, measurements-based methods, and can be either safe or unsafe. A safe method will ensure that the computed value is always greater than the actual WCET value in adding some safety margin to all predictions, whereas other estimation techniques such as probabilistic methods do not provide such guaranty. In one hand, they might be closer to the actual WCET but on the other hand, they cannot guaranty that in practice, the execution time will never be superior to the estimated value. This is problematic for safety-critical real-time systems. Therefore, it is important to know the techniques that have been used in the assessment of a given extra-functional property, as well as the various parameters that these techniques relies upon.

Identified Challenges

From the text above, the challenges that need to be solved to propose a systematic support for the management of extra-functional properties are:

Challenge 1.1 How to support the high heterogeneity of extra-functional properties (hetereogeneity of definition, representation, usage and assessment methods)?

Challenge 1.2 How to ensure that an extra-functional property is used in the intended way?

3.1.2 Extra-Functional Property and Multi-Valuation

During the software development process, extra-functional properties emerge as additional information that needs to be available either to guide the development, to make decisions on the next step to follow, to provide appropriate (early) analysis and tests of the components, or to give feedbacks on the current status. This need for information starts already in early phases of the development, in which extra-functional properties are considered as constraints to be

met and expected to be satisfied later on, thus becoming an intrinsic part of the component or system description.

This implies that through the development process:

1) the meaning of an extra-functional property typically changes from a required property to a provided/exhibited property (see Figure 3.1), and

2) its value changes too as the knowledge and the amount of information about the system increases and as a result of design decisions being settled.

As illustrated in Figure 3.1, extra-functional values are often successively replaced by the latest and most accurate ones. For example, the value of an extra-functional property, estimated in a design phase, is replaced with a new value coming from a measurement after the implementation phase is completed. With more information available, analysis becomes more precise and reliable, and thus is able predict values closer to the actual one.

However, the gradual refinement of an extra-functional property towards more accurate values is not always the expected way to deal with such properties. Often, values which are equally valid in the current development phase, need to exist simultaneously. In other words, this means that the latest value must not replace the previous one.

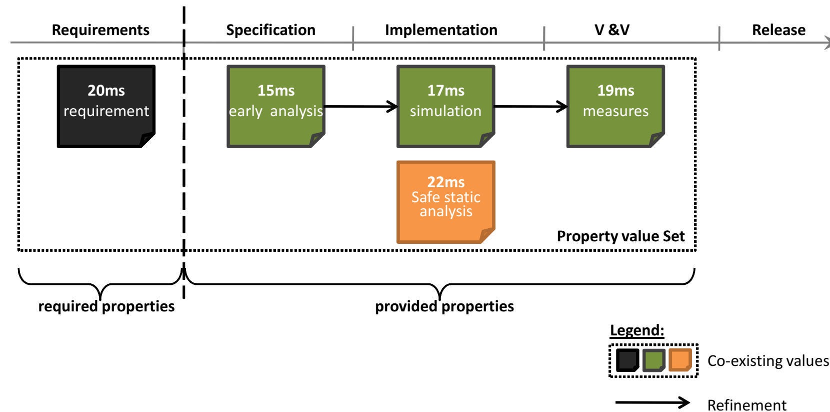


Figure 3.1: Co-existing values for a property

This requires an ability for an extra-functional property to have multiple values to handle different values produced by different sources, to keep the required value and a provided value for verifying the conformity to the initial requirement, or to compare a range of possible values to make a decision. In difference to defining an extra-functional property per assessment techniques such as “measured WCET” or “estimated WCET”, defining an extra-functional property with multiple values allows instead to manipulate the property as a single concept centered around its semantics. This allows to sometimes ignore details related to the assessment methods for example to focus on the value itself. Like this, any of the values of the property could be used as a substitute to perform additional analysis on the system.

Identified Challenges

Following this reasoning, we identify the two following challenges:

Challenge 2.1 How to support the refinement of extra-functional property value during the development process?

Challenge 2.2 How to enable values that are equally valid to co-exist?

3.1.3 Extra-Functional Properties and Reusability

Dealing with extra-functional properties in the context of component-based software engineering also raises the issue of reusability since it is one of the cornerstone concepts around which the component-based approach is built. While efficiency of reuse for the functional part of components has been proven, reuse of extra functional properties is still a challenge.

When a component is reused in different applications or contexts, the extra-functional properties associated to this component must also be reusable, in the sense that their values are still accurate in the current setting. However, many property values depend upon information outside the component model itself. They are for example dependent upon factors such as the overall system architecture, the usage profile, the specific hardware of the targeted platform and even upon the value of other properties.

Therefore in order to reuse the extra-functional properties, means to express the conditions under which the value is correct are required. A typical example is again the WCET, which requires, for a tight result, information about the compiler used to generate the executable code but also about the target platform specification such as the type of memory, processor or the presence of caches, among many other factors.

Hence when a software component is reused in a new context, its corresponding extra-functional properties, assessed in another context, might not be accurate in this one. This means that to keep consistent all the information concerning the component, both its expected behaviour and capabilities and the actual ones, it is necessary to specify the conditions that must be fulfilled so that the value of an extra-functional property remains valid.

However, strictly ensuring the respect of all these validity conditions is a too restrictive approach since in this case, only the values for which the validity conditions are fully satisfied would be reusable. This would limit the reusability of some components only due to their extra-functional property values (values that might be false in the intended context). More practically, a component should be reused even though the values of some of its extra-functional properties are not valid in the new context. In this case, either the extra-functional property should not be reused or it can be still be reused but as a conscious decision of the developer. For example, the value might be reused with a lower accuracy or confidence, or with the data modified to add some safety margins.

Identified Challenges

The questions that emerge here are:

Challenge 3.1 How to represent the context-sensitivity of extra-functional properties?

Challenge 3.2 How to ensure that values are still valid in a new context upon reuse?

3.1.4 Extra-Functional Properties in Hierarchical Component Models

The existence of hierarchical component models that also include composite components — components built out of other components — influences the ways in which the extra-functional properties can be established. Alike the composability challenges for components, we would also like to be able to reason about their composition, in that sense that the values of a property P of a compound element A is the result of a composition of the values of the sub-components $C1$ and $C2$:

$$A = C1 \circ C2 \quad \Rightarrow \quad P(A) = P(C1) \bullet P(C2)$$

with $\circ$ *a composition operator for the components*
$\bullet$ *a composition operator for the properties*

Ideally, all extra-functional properties of a composite component should be directly derivable from the values of its sub-components. However, as described in [28], finding a suitable composition operator for the properties is generally difficult since the value of many extra-functional properties is influenced by other factors such as the software architecture, other properties, the usage profiles and/or the current state of the environment.

Even for composable extra-functional properties, we argue that it is beneficial to allow them to also be stated explicitly for the composite component as such. In particular, this allows analysis of the system also at an early stage of the development when the internals of a composite component under construction are not fully known, or not fully analyzed with respect to the extra-functional properties required to derive a value on the composite component.

The specification of extra-functional properties of a composite component is illustrated by the example in Figure 3.2. The composite component has been explicitly given an estimated value of a static memory usage, and another value is provided by composition, which in this example simply means a summation over the sub-components.

Identified Challenges

The question that derives from above is:

Challenge 4.1 How to support composition of extra-functional property values?

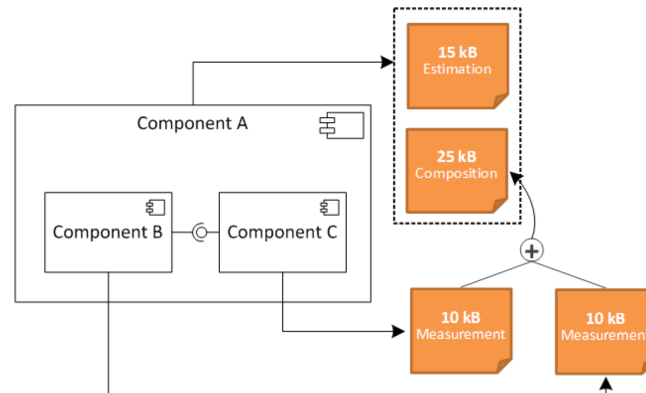


Figure 3.2: A composite component with explicit and derived values.

3.1.5 Extra-Functional Properties and Component Types and Component Instances

Similarly to the object-oriented paradigm, component-based software engineering distinguishes between component types and component instances. A *component type* defines the common characteristics that are shared by all its instances such as the component name, its functionality, the names of its interfaces, its implementation. Conversely, a *component instance* is a representation, either at design-time or run-time, of the corresponding type. Many component instances corresponding to a given component type can be created. Besides, inheriting characteristics from its type, a component instance can also possess instance specific information.

Component instances are used in hierarchical component models to built compound component types, a.k.a. the composites. This is a recursive process, in which a component instance can be in its turn an instance of a composite component type. A representative example of such a case is visible in Figure 3.3 with the instance *B1* of the component type *B*. Indeed, the component type *B* is composite component built out of the instance *E2* of the component type E. As a result, a hierarchical component model leads to have multiple instantiation levels, i.e. a hierarchy of component instances. In spite of this, most component model considers mainly one level of instantiation.

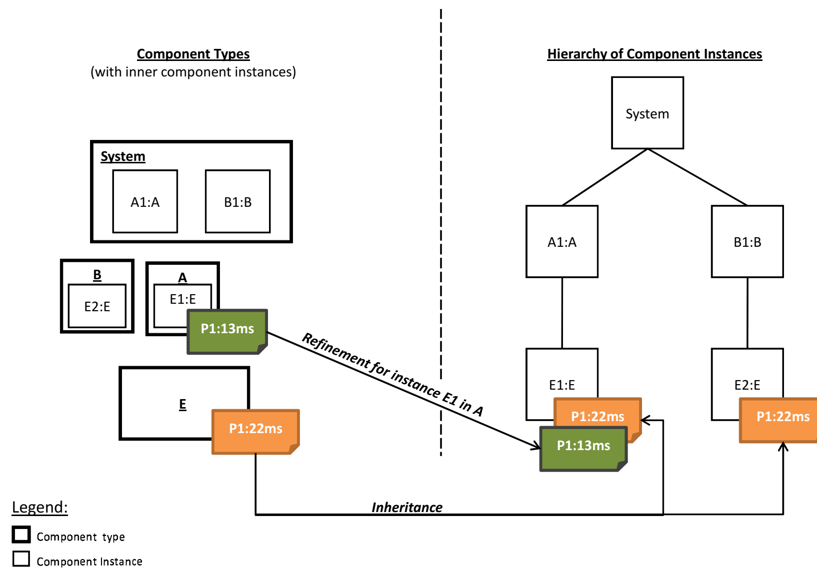


Figure 3.3: Relations between component types and hierarchy of component instances.

When looking at extra-functional properties in this context, the following questions emerge:

1) What are the influences of the dichotomy between component type and component instance on the values of the extra-functional properties?

2) What is the impact of the multiple instantiation levels on these values?

Similarly to the concept of subtyping in object-oriented paradigm, the value of an extra-functional property specified for a component type must also hold for all its instances (see the inheritance link for the extra-functional property P1 between the component type E and its instances in Figure 3.3). However, for some properties, as for example the worst-case execution time, the value of such a property can be smaller in a more constrained environment. Hence in considering the design of the composite component in which an instance is used, extra-functional property values defined on this instance can be make tighter in regards to the values defined on the component type. For example, a smaller value range on an input parameter could remove an execution path that

would otherwise lead to greater WCET. Hence, this means that it should be possible to refine an extra-functional property value, defined for a component type, on one or several of the component instances as illustrated in Figure 3.3 with the refinement link.

However, assessing an extra-functional property value directly on a component type might not be straightforward since this value should hold for all the component instances. Instead, defining an extra-functional property value for a component instance in a given context is simpler. In this case, this means that it should be possible to make a value defined on a component instances available for a component types.

Identified Challenges: ***From Component Type to Component Instances***

Challenge 5.1 What are the extra-functional property values defined on a component type that can be refined on the component instances?

Challenge 5.2 How can this refinement be supported?

Challenge 5.3 Are there any constraints associated with the refinement of a particular property?

Identified Challenges: ***From Component Instances to Component Type***

Challenge 5.4 How does extra-functional property values specified on component instances influence values on the component type?

Challenge 5.5 How can extra-functional property values defined on component instances be generalized to component type?

Challenge 5.6 Are there any constraints associated with the generalization of a particular property?

3.2 Definitions

A straightforward way to specify extra-functional properties is to use $\langle$*name*, *value*$\rangle$-pairs of annotations. However, this gives too much freedom concerning the definition and it brings problems to manage extra-functional properties at a large scale or in automated processes such as composition or analysis.

In order to move towards a precise formalisation of extra-functional properties, which allows an unambiguous understanding and a precise semantics both with respect to meaning and valid specification format of the value, we consider extra-functional properties as multi-valued and context-aware artefacts that must be integrated into component models and managed in a systematic manner. Accordingly, in this section, we define the concept of multi-valued context-aware extra-functional property through a set of formal definitions. These definitions are the foundations for the development of a framework for integrating extra-functional properties in component models and managing them in a systematic manner.

This framework is based upon two formal definitions: the definition of *attribute type* that specifies a class of extra-functional properties and the one of *attribute instance* (also called attribute value) that refers to a given extra-functional property value associated with a specific element of a component-based design. This is similar to the dichotomy between the concepts of "class" and "class instance" in object oriented programming. Likewise, an attribute instance must comply with the specific structure imposed by its corresponding attribute type.

Notations

We denote by $F(e)$ (resp. $G(e)$), the function that retrieves the element F (resp. G) from a tuple $e = \langle F, G \rangle$.

3.2.1 Attribute Type

The attribute type provides a consistent definition for the representation and usage of extra-functional properties. It specifies how a given extra-functional property is represented, i.e. what data type is required for its values and how they should be manipulated. Having such a definition serve as basis to automate the assessment of extra-functional properties during the development process.

Definition 1. An attribute type is defined by a tuple Att_{type} so that:

$$
\begin{aligned}
Att_{type} \quad = \quad & \langle TypeID, Attributable^{+}, Data_format, \\
& SupportMechanism, Documentation \rangle
\end{aligned}
$$

where,

- $TypeID$ is a unique identifier for the type.
- $Attributable$ is a set the elements of a component model to which extra-functional properties of type TypeID can be attached to.
- $Data_format$ specifies the data type used to represent the values.
- $SupportMechanism$ is a tuple specifying mechanisms to manipulate the extra-functional properties in a consistent way.
- $Documentation$ describes the extra-functional properties in natural language. That documentation must supply enough information to primarily clarify the meaning of the attribute type as well as its intended usage.

Type Identifier

The type identifier element (i.e. $TypeID$) is the key that allows retrieving the corresponding attribute type. For simplicity purpose, the name of the property is used as the unique identifier in the remaining of the thesis as illustrated in Table 3.1. Table 3.1 gives an illustrative representation of some attribute type specifications.

Attributable

As mentioned in [7], the additional information provided by attributes does not necessarily concern the component as a whole, but in fact often points more precisely to some parts of a component such as an interface or an operation of an interface. In our view, this relation should not be limited to components, interfaces and operations, but be extended so that attributes can be

associated with other elements of a component model, including for example ports, connectors or more notably component instances. For instance, having an extra-functional property on connectors to capture communication latency makes it possible to reason about the response time of complex operations that involve communication between components. Similarly enabling specifying extra-functional property on component type and component instance is the first step towards enabling their refinement as envisaged in Section 3.1.5.

Following this standpoint, we define as *attributable* an element of a component model (*component*, *interface*, *component instance*, *connector*, etc.) to which extra-functional properties (*attributes*) can be attached. By this means, all attributable entities are treated in similar way with regards to the definition and usage of attributes. $Attributable$ hence represents the set of the elements of a component model to which extra-functional properties of a given type $TypeID$ can be attached to.

Data Format

The set of possible data format varies a lot from one property to another as explained in section 3.1.1. This means that the part of attributes concerned with expressing data must be represented in an unambiguous and well-tailored format implying that in addition to supporting primitive types such as integers, floats, etc., and structured types such as arrays, complex types must also be covered. These complex types include representation of value distributions, various external models, images, formulas, etc. The data format is defined through a data type that describes the precise storage format that an extra-functional property value must conform to. The data type must be issued from a type system.

Support Mechanisms

Support mechanism corresponds to the collection of all the mechanisms required to handle extra-functional properties in a consistent way. Such mechanisms include, but are not limited to, suitable compositional operators, viewers, editors, refinement policies, allowed cardinality, etc. In that sense, the set of supporting mechanisms is defined as follows:

$$
\begin{aligned}
SupportMechanism \quad = \quad & \langle Cardinality, Operator, Policy^{*}, \\
& Viewer, Editor, \ldots \rangle
\end{aligned}
$$

where,

- $Cardinality$ represents the cardinality of the extra-functional properties, i.e. how many values an instance of this attribute type can have for a given attributable.
- $Operator$ specifies the compositional operator if any, i.e. how to derive the value of an extra-functional properties specified for a composite component from the sub-components and from the environment.
- $Policy$ is one of the refinement policies described in Chapter 4.
- $Viewer$ specifies how the extra-functional properties should be visualised.
- $Editor$ specifies how the properties should be modified.
- $\ldots$ informally denotes that additional supporting mechanisms could exist in the tuple.

3.2.2 Attribute Registry

Each attribute type is stored in a repository of attribute types, which contains the pool of extra-functional properties that can be assigned to the entities of component models, i.e.:

Definition 2. An $attribute\ registry$, $\mathcal{R}$, is a set of all attribute types available in a given design context or in the supporting developing environment.

Table 3.1: Attribute type specification (without documentation).

TypeID	Attributables	Data Format	Examples of Support Mechanisms
Value Range	Port	[Float, Float]	Viewer: *Values are visualized directly.* Editor: *Values are modified with a dedicated editor.*
WCET	Component, Instance, Service	Int	Viewer: *Values are visualized directly.* Editor: *Values are modified directly.* Policy: *Values can be refined in the component instances.*
Static Memory Usage	Component, Instance	Model	Viewer: *Values are visualized with a dedicated viewer.* Editor: *Values are dedicated editor.* Operator: *Values for composite components can be derived by adding the values of the component instances.*

In this context, the uniqueness of each attribute type must be ensured as implied by Property 1 below.

Property 1: *Each attribute type in the registry $\mathcal{R}$ is unique, i.e.:*
$$\forall a_1, a_2 \in \mathcal{R},\ if\ \mathit{TypeID}(a_1) = \mathit{TypeID}(a_2)\ then\ a_1 = a_2.$$

Although this way of specifying attribute types provides the great advantages of being open and extensible so that it can fit the multitude of extra-functional properties which need to be defined, it still requires users to have an intuitive and common understanding of what the meaning and intended usage of the attributes were when they were created. Therefore, it is important to provide proper attribute type *documentation*.

If the repository of attribute types is global, that is, it contains all possible extra-functional properties independent of an application, it is then reasonable to assume that hundreds of attribute types or more will be stored in it. Several classification schemes (e.g. [66] and [67]) have been proposed which can be used as basis to identify groups of attribute types such as “resource usage”, “reliability”, “timing”, etc. These categories could allow navigation across attributes more easily and possibly hide the whole subset of attribute types that are uninteresting for a particular project. A remaining challenge is in this case to determine appropriate categories, as the proposed classifications are distinct and often non-orthogonal as mentioned in [28]. However, this is not within the scope of the definition.

3.2.3 Metadata Type

It is important to document the way the value of an extra-functional property has been obtained to ensure that information about a component (or another element of a component model) is correct and up-to-date, also when the component is reused. This is done through the concept of attribute value metadata, or simply metadata, which role is to capture the context in which the corresponding attribute value has been obtained: when, how and, possibly, by whom and in which context.

Similarly to attributes, the concept of metadata also distinguishes between *metadata type* and *metadata instance*, where metadata type specifies the commonalities shared by all the instances of a given type, and a metadata instance is simply a value of the metadata, which characteristics conforms to what the type imposes. More formally, a metadata type is specified as follows.

Definition 3. A metadata type is defined by a tuple M_{type} so that:

$$M_{type} = \langle MetID, Metadatable^{+}, Value_format, Cardinality, \ldots \rangle$$

where,

- $MetID$ is a unique identifier for the type.
- $Metadatable$ is a set of attribute TypeIDs to which the metadata of type MetID can be attached to.
- $Value_format$ specifies the type used to represent the values.
- $Cardinality$ represents the cardinality of the metadata, i.e. how many values of this metadata type a given metadatable can have.
- $\ldots$ informally denotes that additional supporting mechanisms could exist in the tuple. For example, it can be specified whether a metadata must be present for a given metadatable.

Table 3.2 lists some examples of metadata type specification. However, the question of determining a complete and non-orthogonal list of metadata types that must be specified remains.

Alike attribute types, metadata types must be stored in a repository. This repository of metadata types must have access to the repository of attribute types to be able to link the metadatable to existing attribute types.

Table 3.2: List of possible metadata type specifications in which "*" means "any number of"

MetadataID	Desc. Of	Value Format	Card.	Example of an Additional Mechanism
Creation Time	*	Timestamp	1	Mandatory: *All attributes must have this metadata*
Modification Time	*	Timestamp	1	Mandatory: *All attributes must have this metadata*
Version	*	Int	1	Mandatory: *All attributes must have this metadata*
Accuracy	*	Float	1	Optional: *Can be added on demand*
Type	WCET BCET ACET	{"Estimation", "Guarantee" }	1	Mandatory: *All the attributes which type is in "DescriptorOf" must have this metadata*
Source	*	{"Estimation", "Measurement", "Simulation", "Derived", . . . }	*	Optional: *Can be added on demand*
Comment	*	Text	*	Optional: *Can be added on demand*

3.2.4 Attribute Instance

Values of extra-functional properties are defined as attribute instances that conform to an attribute type Att_{type} represented by its unique $TypeID$. Providing that it is authorized by its type specification, an attribute instance can be associated with any entity of a component-based design such as component, service, port, connection or even component instance.

Definition 4. An attribute instance Att_{inst} is defined by:

$$Att_{inst} = \langle TypeID, Data, Metadata^{+}, ValidityConditions^{*} \rangle$$

where

- $TypeID$ is the identifier of the corresponding attribute type.
- $Data$ contains the concrete value for the property. The type of the data must conform to the data format specified in the corresponding attribute type.
- $Metadata$ is a set of metadata instances represented as $\langle name, value \rangle$ pairs.
- $ValidityConditions$ describes the conditions under which the value is valid.

Notations

We denote by $Att_{inst}(e)$, the set of the attribute instances attached to an element e of a component-based design $\mathcal{CM}$, and by $att_{inst_t}(e)$ a set of the instances so that $TypeID(att_{inst_t}) = t$. Additionally, for an instance $att_{inst_t}(e)$ of an element $e \in \mathcal{CM}$, we use $|att_{inst_t}(e)|$ as the notation for the number of instances of type t attached to e.

Data

Data contains the concrete value for the attribute instance. Its type must conform to the data format specified for the corresponding attribute type. For example, for the attribute type "*Value Range*" defined in the repository shown

in Table 3.1, the data format is specified as an interval of two rational numbers. Accordingly, a correct value for the data of a corresponding instance could be [*10.5, 20.3*]. Formally, this is specified by the following property:

Property 2: *For an attribute instance $att_{inst}(e)$ of an element e of a component-based design $\mathcal{CM}$, the type of its data d is conformed to the data format specified in the corresponding attribute type:*

$$\forall att_{inst_t}(e) \in Att_{inst}, \quad type(d) = Data_format(att_{type_t})$$

- *where, $type$ is a function allowing to retrieve the type of a data.*

Metadata

Metadata are instances of a given metadata type as specified in Section 3.2.3. A metadata instance M_{inst} is simply defined as a identifier-value pair: $M_{inst} = \langle MetID, Value \rangle$. Similarly to what is specified in Property 2 for the data of attribute instance, the type of the value of a metadata instance must also conform to the value format specified in the corresponding metadata type.

Validity Conditions

In studying the characteristics of extra-functional properties in component-based development (described in Section 3.1.3), the following additional concern emerges:

> *Data of a given attribute instance are not necessarily valid in all context.*

Accordingly to fully benefit of extra-functional properties during component-based development, it is important to know in what context an attribute instance, or more exactly its data, can be used or not. This is the role of the validity conditions which explicitly describe the conditions in which an attribute value can be trusted. Validity conditions can be seen as a set of restrictions of the applicability context of the attribute instances. Constraints on the underlying platform, specification of usage profile, and dependencies towards other attributes are examples of such conditions. However, many different conditions can be defined and, as with attribute types, an attempt to identify them all is bound to fail. Yet, validity conditions must be defined in a strict manner and it is important that they are publicly exposed.

We identify three main "types" of validity conditions that must be expressed:

- *Always*: the value of the attribute instance is applicable in all context, i.e. the value of the property is context-independent. A typical example of such property is "Line of code".
- *A set of conditions*: the value of the attribute instance is guaranteed to be valid when the current context of use matches all the conditions. For example, extra-functional properties such as "response time", "memory usage" which values are tightly dependent upon the target platforms and the manner in which they have been assessed would need to have conditions such as "*Platform*="Lego Mindstorms RCX" **AND** *Source*="Static Analysis with BoundT" " defined (in this particular example, platform and source refer to the two corresponding metadata types).
- *Unknown (or undefined)*: there is no context associated with the attribute instance. No guarantee can be made on its accuracy.

The complete definition on how validity conditions must be expressed remains to be done.

3.3 Summary and Discussions

In this chapter, we have started by identifying a number of challenges resulting from the use of extra-functional properties in component-based development. These challenges relate to the heterogeneous nature of extra-functional properties (Challenges 1.1 and 1.2), their purpose within the development (Challenges 2.1 and 2.2), and other specific aspects directly related to the key principles of component-based software development that are reusability (Challenges 3.1 and 3.2), composability (Challenge 4.1) and the relations between component type and component instances (Challenges 5.1 to 5.6).

To address these challenges, we have then introduced the concept of multi-valued context-aware extra-functional properties that highlights two important aspects of the use of extra-functional properties in component-based development:

1) their multi-valued nature, that is several extra-functional property values can be equally valid in a given development context and therefore must co-exist, and

2) their context-awareness, i.e. extra-functional property values are typically dependent upon their usage context and this dependence must be captured and made explicit in order to facilitate reusing the extra-functional properties together with the component they describe for example.

Accordingly, the concept of multi-valued context-aware extra-functional properties is formally defined through the provision of four key definitions, namely attribute type, attribute instance, attribute registry and metadata type. Altogether, this builds the foundations towards the systematic specification, management and integration of extra-functional properties in component-based development.

Chapter 4

Managing Multi-Valued Context-Aware Extra-Functional Properties

In introducing, in Chapter 3, the concept of multi-valued context-aware attributes to specify extra-functional properties, several challenges arise. These challenges are mainly inherited from the ability for each attribute to have multiple values that are possibly equally valid in the current development context. These values have been assessed in different ways, using different assessment methods for example, or they have been assessed in different contexts. This leads to the possibility to have a large number of values from which it is necessary to find the most suitable values to use in a given development context. When the number of values goes above a certain threshold, this amount of data actually becomes an hindrance. This can lead to an increase of the time needed to identify relevant values or, in the worst case, to completely fail to spot them. In that context, it becomes necessary to determine proper supporting mechanisms to alleviate this issue and facilitate the management of extra-functional properties. Accordingly, the purpose of this chapter is twofold:

- Precisely identify the challenges brought by the introduction of multi-valued context-aware extra-functional properties.
- Investigate and develop possible solutions to facilitate the management of multi-valued context-aware extra-functional properties.

4.1 The Inherent Challenges

Due to the introduction of multi-valued context-aware extra-functional properties, challenges emerge to enable their management during software management. These challenges are the following:

Redundancy

Extra-functional property values being estimated by different assessment techniques are generally different. Yet, nothing prevents a value to be produced multiple times. For example, such redundant values appear when the same analysis is applied several times but also when applied on different platforms although the value of the extra-functional property is actually platform independent.

Redundancy is a problem often encountered in database management that leads to engineering and information overhead. In that particular domain, normalisation techniques are used to remove redundant values while preserving data integrity. Similar solutions are needed in the management of the redundancy of multi-valued context-aware extra-functional properties. However, the definition of such mechanisms requires first to be able to identify the redundant values before deciding how to handle them.

Applicability

As explained in Chapter 3, extra-functional property value can be assessed in different context with possibly different validity conditions. In that context, the question of the pertinence of the value in the current development context arises. For example, a worst-case execution-time analysis has been performed for a component on a given platform. Later on, this component is intended to be reused on another platform. In that scenario, the component integrator faces three choices:

1) This value is useful in the current development context.
2) This value is not at all applicable in the current development context.
3) This value is not directly applicable in the current development context but it would be interesting to use it anyway, as an early estimation to perform some analysis for example.

As a consequence of this possibilities, mechanisms to handle the diversity of extra-functional property values must be provided.

Confidentiality

In enabling extra-functional properties to be described through the provision of suitable metadata and/or the context under which the value has been obtained, this also allows to integrate the specification of functional properties without hampering the utilisation of interfaces. In this context, functional properties do not refer to interface specification of the operations handled by the components, but to the modelling of the behaviour of the components in a format suitable for analysis techniques such as timed automata model. By this means, our intention is to increase the analysability and predictability of component-based embedded systems, and enabling a seamless and uniform integration of existing analysis and predictions theories into component models. However, this reveals information concerning the details of the implementation of the system or the components. This is not a major issue for in-house development, but it naturally becomes more problematic for its utilisation in the development of systems or components for which the implementation details must remain hidden such as COTS components. In that context, all the models that have served for analysis are packaged together with the components. The question that arises in that case is how to ensure that the use of multi-valued context-aware extra-functional properties does not reveal confidential information.

4.2 Identified Supporting Mechanisms per Management Concerns

In order to address the redundancy, applicability and confidentiality challenges described in the previous section, the following management concerns must be considered: conciseness, relevance, accuracy, transparency, consistency. In this section, for each of this management concern, we identify a set of supporting mechanisms that are necessary to efficiently manage multi-valued context-aware extra-functional properties in component-based development for embedded systems.

Conciseness

Except for specific cases, having the same value of an extra-functional property multiple times is not more informative than having this value only once. On the contrary, this is counter-productive because it adds information overhead for the users. The conciseness principle consists in displaying only strictly different values whenever possible. Conciseness can be attained by :

- *Value Hiding*. Value hiding is concerned with hiding multiple versions of a value: only one version for each variant of attribute value should be displayed. Other versions should be retrieved on demand.

- *Value Aggregation*. When several occurrences of a value can be found, these values can be merged together including their metadata and validity conditions into a single value.

- *Duplicate Removal*. Strictly redundant values should be removed.

Relevance

In a given development context, not all the extra-functional properties attached to an element of the design are relevant. Only a subset of the values accurately describes that element. In order to limit the engineering overhead and facilitate decision making, only the values relevant in the current context should be visible and additional values should be accessible upon request when needed. Relevance can be attained by:

- *Value Filtering*. Value filtering is closely related to value hiding but with the difference that value filtering aims at masking irrelevant values in the current context. That is to say values that are not directly related to the current development context should be filtered out. To do so, it is necessary to establish first the criteria that are required to identify the values to filter out and then to determine what to do with the values that are not directly relevant in the current context but that the developer explicitly want to have.

Accuracy

Having relevant extra-functional property values is not enough. They also must be accurate. That is, they must describe as closely as possible the element they are attached to. For example, values of extra-functional properties contributing to a component type are often over estimated to ensure that they are still valid when the component is instantiated in several different contexts. Yet, in benefiting for the knowledge of the usage context, values can be made tighter. Accordingly, accuracy can be attained by:

- *Value Refinement*. Value refinement is the process of gradually revising an extra-functional property value during the development process towards its most accurate quantity. We differentiate between two types of value refinement: *refinement over time* when the refinement is done during the development as the knowledge of the system increases, and *refinement between component types and component instances*.

Transparency

In general, available tools to compute extra-functional properties do not know how to handle multiple values as they generally assume a single value such as the WCET analysis presented in [65] which considers only one WCET value for each component. Accordingly, since most analysis methods can only process one value, the questions that arise are: 1) which values are of interest for a particular analysis, 2) how to select and use these values only and 3) how to ensure that only one value is available per element. Transparency can be attained through:

- *Value Selection*. This enables providing a suitable value to use in selecting the most representative one in the current context according to a given set of criteria. Such mechanism would facilitate integration of existing assessment methods for extra-functional properties and their automation in the development process. Value selection is dependent upon the task accomplished by an user.

Consistency

Maintaining data consistency is a well-known problem in data management. During the development process, artefacts used in the assessment of extra-functional property values are often created, modified, moved or sometimes even removed. This might lead a situation in which information provided by an extra-functional property is not consistent with the supporting artefacts. In practice, a lack of consistency management leads to lower the confidence in the available extra-functional property values and force designers to recompute and reassess the values after each modification or when a component is reused.

In addition, many extra-functional properties are co-dependent, i.e. the value of an extra-functional property directly influences one or several other extra-functional property values. This means that a change in one of the value should be reflected in the dependent values to keep them consistent.

As a consequence, it important to ensure that the extra-functional properties and the related artefacts are kept consistent. Consistency can be attained by:

- *Consistency Constraints*. This mechanism ensures that rules specifying relationships between extra-functional property values are specified are enforced.

- *Value Evolution Tracking*. The purpose of evolution tracking is to identify changes that have occurred during the development process that influence an extra-functional properties value. Two different types of evolution tracking can be distinguished: *i*) *tracking evolution of refined values*, that enables identifying changes that have been performed of on a value and checking whether the new values conforms to particular criteria. *ii*) *tracking evolution of dependencies*, that allows to detect changes in elements upon which an extra-functional property value depend (implementation, analysis model or another property for instance). In order to enable value evolution tracking, mechanisms to enable managing traceability between the source artefacts of the analysis and the assessed values must be present. More details on a solution to address this issue can be found in [68].

4.3 Two Supporting Mechanisms

In this section, we describe two of the identified supporting mechanisms, namely value selection in Section 4.3.1, and value refinement between component type and instances in Section 4.3.2.

4.3.1 Value Selection

In introducing the possibility to have a different value for each specific usage context, the number of possible configurations of attribute values to consider can become too big to be manageable by hand. In addition, many tools assume a single extra-functional property value for each element. This is why an automatic or semi-automatic selection mechanism should be available for the users.

To be able to select relevant values, it is necessary to be able to tell them apart. To do so, we adopt similar principles to Software Configuration Management (SCM) for the management of multi-valued context-aware extra-functional properties. As explained in [69], SCM distinguishes two types of versioning elements:

- **Versions** (also called revisions) that identify evolution of an item over time. Usually the latest version of an item is the one available by default, but an older version can also be used instead, for example through a timestamp to select the latest version created before a specific time.

- **Variants** which allow the existence of different versions of the same item at the same time.

These two concepts can directly be applied to the management of multi-valued context-aware extra-functional properties, thanks to the possibility to have multiple values for attribute instances and the presence of metadata that allows distinguishing them.

Accordingly, a subset of attribute instance (versions or variants) can be obtained by the use of appropriate *matching conditions*. A matching condition is a statement that is either derived from the set of available metadata types or taken from a list of predefined keywords as specified in Definition 5.

Definition 5.
Formally, we define a matching condition as:

$$
\begin{aligned}
\langle \text{Condition} \rangle &::= \langle \text{MetCond} \rangle \mid \langle \text{KeyCond} \rangle \\
\langle \text{MetCond} \rangle &::= \langle \text{MetID} \rangle \, \langle \text{Op} \rangle \, \langle \text{Value} \rangle \\
\langle \text{Op} \rangle &::= \text{“}=\text{”} \mid \text{“}\neq\text{”} \mid \text{“}<\text{”} \mid \text{“}\leq\text{”} \mid \text{“}>\text{”} \mid \text{“}\geq\text{”} \\
\langle \text{KeyCond} \rangle &::= \textit{a set of predefined keywords} \\
\langle \text{MetID} \rangle &::= \textit{existing metadata type identifiers} \\
\langle \text{Value} \rangle &::= \textit{values}
\end{aligned}
$$

For example, any metadata type specified in Table 3.2 in Chapter 3 can serve as a basis to create matching conditions such as *Platform* = “*ARM7*”, *Source* = “*Estimation*”. Keywords, such as *latest* to get the most recent version or *before* to obtain value created before a certain timestamp, can be added as additional selection conditions and shorthand notation for the most commonly used matching condition and their combinations (see “and”-conditions in selection filter below).

From the selection point of view, metadata and validity conditions are equivalent. In the selection process, the configuration filter defines constraints over metadata or validity conditions in the same way. The difference is however in understanding the purpose behind the use of the configuration filter and in helping the developer in detecting possible problems in its definition.

A *configuration filter* enables to have more control over the values to retrieve by using one or several matching conditions. It is formally specified in Definition 6. As illustrated in Figure 4.1, a configuration filter can be seen as a sequence of matching conditions, combined through *AND* or *ELSE* connectors. The “else”-conditions are tested in order until a subset of attribute instances is selected. The “else”-condition *(1)* is evaluated first. If there is no attribute value corresponding to the matching condition, then the second “else”-conditions *(2)* is examined, and so on until either values are found or there is no value that corresponds to the configuration filter.

Definition 6.
Formally, we define a configuration filter as:

$$
\begin{array}{rcl}
\langle \text{Filter} \rangle & ::= & \langle \text{ConditionOr} \rangle \ | \ NULL \\
\langle \text{ConditionOr} \rangle & ::= & \langle \text{ConditionAnd} \rangle \\
 & | & \langle \text{ConditionAnd} \rangle \ \textbf{ELSE} \ \langle \text{ConditionOr} \rangle \\
\langle \text{ConditionAnd} \rangle & ::= & \langle \text{Condition} \rangle \\
 & | & \langle \text{Condition} \rangle \ \textbf{AND} \ \langle \text{ConditionAnd} \rangle
\end{array}
$$

$$Condition_1 \ \textbf{AND} \ Condition_2 \quad \textbf{ELSE} \qquad (1)$$

$$Condition_3 \ \textbf{AND} \ Condition_4 \quad \textbf{ELSE} \qquad (2)$$

$$\vdots$$

Figure 4.1: Abstract Representation of a Configuration Filter

A configuration filter can be applied to the entire system, or to a set of components, and then all architectural elements expose particular versions of the attributes that match the filter. This is important when some system properties are analyzed using consistent versions of several attributes (for example in an analysis of a response time of a scenario performed on a particular platform). For example, it is possible to define a configuration filter to apply on the components in Figure 4.2. This configuration filter should *"select attribute values that have been assessed by measurement for platform 'X', or, alternatively, values which have been defined for the release 2.0. In case no value corresponding to these criteria can be found, it is possible to select the latest values"*.

In this example, the configuration filter is expressed as follows:

(*Platform* = *"X"*) **AND** (*Source* = *"Measurement"*) **ELSE**

(*Label*= *"Release 2.0"*) **ELSE**

Latest

First, it will attempt to select first all the values which metadata matches "*Platform*= '*X*' " and "*Source*= '*Measurement*' ". If no value corresponds to these two conditions, the filter will try to find values with a Label metadata with a "*Release 2.0*" value. Again, if no value can be found, the latest version of each attribute instance value is retrieved. The selected values are marked with a tick in Figure 4.2.

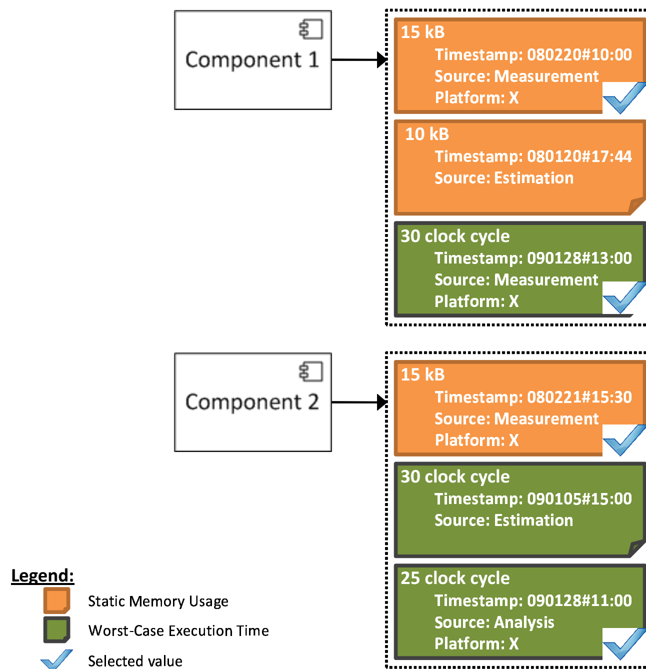


Figure 4.2: Attribute value selection.

In practice, two possibilities of dealing with attribute values exist:

- **Attribute navigation.** The possibility to navigate explicitly through different versions of an attribute (i.e. through different values), and update the selected value (changing data, or metadata information, or modifying the validity conditions).

- **Configuration.** Values are selected, for one or several attributes, according to a given selection principle (e.g. based on version name or timestamp).

The first method is intended to be interactive and requires manual intervention from the users. The underlying purpose is that the environment should provide users with minimal set of values to view (see the conciseness principle in Section 4.2). By default, this minimal set of values can be obtained by having only the latest values available. However, it is important to enable users to easily browse and access the "hidden" values when needed. On the other hand, the goal with the second method, i.e. the configuration method, is to have automated activities. This can be useful for example to automate analysis that must be based on specific extra-functional property values.

The selection mechanism described here does not ensure the uniqueness of the selection per element: i.e. that only one value is selected. If the configuration filter leads to the selection of multiple values, several options exist according to the purpose of the configuration filter. For attribute navigation, the latest version is selected by default. For automated activities, several options must be made available:

- *manual selection*: let the user decide the value to use wherever multiple values have been retrieved;

- *automatic selection*: a warning is logged and a non deterministic selection is performed on this value set. However, the user must have the possibility to review the selected values and change the selections manually.

Moreover, for automated activities, the selection mechanism does not guarantee the existence of a value for all existing attribute instances. However, in suitably configuring the selection filter, equivalent attribute values can be used as substitutes of a value which otherwise would be missing. If no equivalent value can be found, an error must be raised.

4.3.2 Value Refinement between Component Type and Instances

In this section, we introduce the concepts and mechanisms to enable refining multi-valued context-aware extra-functional properties in hierarchical component models between a component type and its instances. This is usually not supported in component models. The first step to enable such a refinement is to explicitly specify the relationships that are allowed between a type and its instances with respect to the attribute values. This is done through the definition of a metamodel that precisely specifies these relations (see Figure 4.3). Additionally, explicit definitions of property inheritance and refinement policies are also needed. These policies formally specify consistency constraints between the refined values and the original ones. Without such policies, the consistency between refined values and the original ones cannot be ensured.

Inheritance Refinement Metamodel

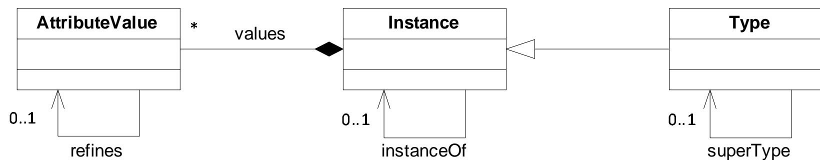


Figure 4.3: Metamodel for Multi-Level Instantiation and Refinement Support of Extra-Functional Property Values

The metamodel shown in Figure 4.3 describes the key concepts we propose to enables a multi-level instantiation of extra-functional property values with a support for their refinement. In it, the metaclass *AttributeValue* represents the attribute instances as defined in Chapter 3. The possibility for an element to have multiple attribute values is enabled through the composition links *values* between the two metaclasses *AttributeValue* and *Instance*. The metaclass *Instance* refers to any element which is an instance of another element such as a component instance. The metaclass *Type* represents object type such as a component type. Furthermore, object type can also have multiple attribute instances thanks to the inheritance link between the *Instance* and *Type* metaclasses. To allow consistency checking, refinement between values must be tracked. This is done through the *refines* relationship.

Refinement Mechanisms

To make the approach as generic as possible, we use the type instantiation and specialization paradigm to support the refinement of extra-functional property values. An object (component or any model element) can be refined by creating a new object which is an instance of the original one (e.g. a component instance). We choose the following definition for a refined object: *"an object is a refinement of another object if all information defined by the original object is still valid for the refined object"*, i.e. an original object is an abstraction of a refined object. Furthermore, several objects can refine the same original object, hence creating multiple variants of this object. These refined objects can in their turn also be refined.

As illustrated in Figure 4.3 and described below, two mechanisms of refinement are provided: *refinement by instantiation* and *refinement by specialization*. These mechanisms are based on the following assumptions:

- *Assumption 1*
  Extra-functional properties are defined as annotations on the model elements;
- *Assumption 2*
  Multiple values of extra-functional properties can be defined and there are means to distinguish between them (using metadata for example);
- *Assumption 3*
  An extra-functional property value is associated to exactly one model element;
- *Assumption 4*
  A refined object must be attributable, i.e. it should be able to have its own extra-functional property values;

We distinguish between two types of refinement:

1) *Refinement By Instantiation*

   The type-instance design pattern is often used in modelling languages to allow specifying information in the type that will be shared by a set of objects, i.e. the instances. There is an implicit conformity between the instances and their type. For example, object-oriented programming languages rely heavily on this pattern in which a class defines the set of attributes and methods that all object that are instances of this class will

inherit. In such languages, conformity is checked at compilation time and at runtime. In general, an instance cannot be a type, which limits the number of instantiation levels to one.

In our case as explained in Section 3.1.5, we want to allow refining an object as many times as necessary. In this case, the number of instantiation level is not limited. That is why a *Type* inherits from *Instance*. It becomes possible to have instances which are also types enabling refining them with their instances.

To have explicit refinement traces, an instance is linked to its type thanks to an *instanceOf* link. In order to facilitate evolution management, we choose to forbid an instance to change its parent after creation time. In other words, the *instanceOf* link destination is defined at the creation time of the source element.

2) *Refinement By Specialization*

In object oriented languages, a class can be the specialization of zero, one or many other classes. A child class inherits all information from the parent ones except some of them (for example their names). The child class refines its parent class by adding new information such as new attribute and new methods. We choose to manage only simple inheritance where a class can at most inherits from another class. To have explicit refinement traces, a type is linked to his parent type thanks to a *superType* link. As with instantiation, we choose to forbid a subtype to change its super type, i.e. to point to another type, after creation time.

Inheritance Policies

The computation of refined attribute values is guided by inheritance policies to ensure the consistency between refined values and the original ones. We have defined three inheritance policies, that are *Final*, *Override* and *NotInherited*. An attribute type set with a final inheritance policy implies that that attribute instances corresponding to this type and defined on a component type will always be inherited on the component instances. However, the values cannot be modified on the instances; they can only be modified on the original object on which it has been defined. An override attribute type is similar to a final attribute type with the difference that inherited values can be modified on the instances. In that sense, the value can be refined. In that case, OCL constraints

can be specified to check the consistency of the refined value with the parent one. Finally, a notInherited attribute value is never inherited.

Table 4.1 gives some examples of possible attribute inheritance policies for various extra-functional properties. The attribute type “Vendor Name” cannot be inherited at all. As a consequence, there is no need to have refinement constraint defined for this attribute type. Conversely, the other attribute types can be refined. As an example, if the WCET attribute type is specified as override with the constraint that the refined value cannot be greater than the original value, this definition will guarantee that all refined WCET values should be smaller than the original value.

Table 4.1: Examples of attribute inheritance policies.

Identifier	**Inheritance Policy**	**Constraint**
Vendor Name	NotInherited	None
WCET	Override	OriginalValue $\geq$ RefinedValue
Static memory	Final	None

Illustrative Example

Figure 4.4 illustrates an example of the creation of an instance of the model element $O1$. All contained elements, i.e. the transitive closure of containment, which includes $O2$ and $O3$ model elements, are instantiated together with $O1$. $O5$ which is not contained but referred to is not instantiated. All the links pointing to O5 are instead cloned on the newly created instances: in this example, $O3'$ is linked to $O5$.

Taking the inheritance policies for the attribute type defined in Table 4.1 as example, the VendorName value is not available on the instance $O1$ whereas WCET and static memory are available. This example also shows that the WCET value has been refined to a smaller value in $O1$ and the static memory usage cannot be changed since it is defined as final.

A illustrative example based on the proposed refinement mechanisms done on a concrete component model is proposed in Chapter 8.

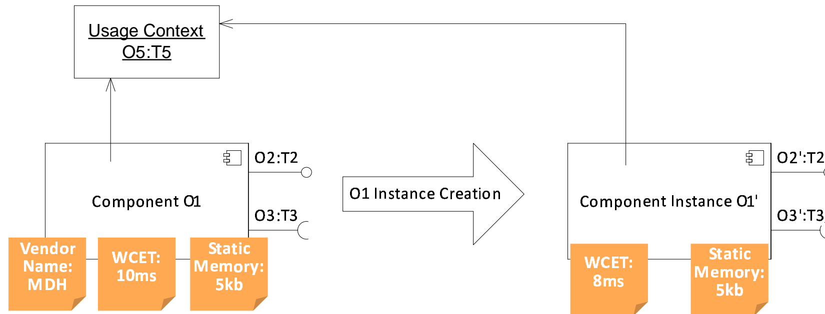


Figure 4.4: Illustration of Instantiation with Extra-Functional Property Values

4.4 Summary

In this chapter, we have started by identifying three main challenges that derive from the use of multi-valued context-aware extra-functional properties. In order to provide a seamless and efficient management for these properties, suitable supporting mechanisms must be defined. Accordingly, we have first specified a set of basic supporting mechanisms to address the aforementioned challenges, before proposing approaches for two of them; one for the selection of attribute values corresponding to particular criteria and another one for handling the refinement of attribute values between component type and instances.

According to how configuration filters are expressed and on what element they are applied, the selection mechanism can be used for both hiding values and selecting specific values. The values can be hidden or selected based on their metadata, validity conditions, and other commonly used filtering criteria such as latest.

In addition, the selection mechanism facilitates automated analysis in allowing the use of semantically equivalent extra-functional properties when a value is a missing value. For example, an analysis technique such as "response time analysis" that uses the worst-case execution time property values defined on services can used indifferently a worst-case execution time that is obtained by measurements or by static safe analysis.

The approach to refinement of property values between component types and instances also facilitates the uses of equivalent values. In particular, it avoids having to reassess values to make early estimations.

Chapter 5

nLight — The Attribute Framework

Based on the concepts presented in Chapters 3 and 4, we have implemented nLight, a framework that supports the management of multi-valued context-aware extra-functional properties. The main purpose of this framework is to provide a uniform and user-friendly structure to seamlessly and systematically specify, integrate and manage extra-functional properties in component-based development. In its current implementation, it is built as a set of Eclipse plugins using the Eclipse Modeling Framework and it is intended to be used conjointly with any development tool built on top of a component model[1] defined through a metamodel. This allows extra-functional properties to be easily attached to selected types of architectural element of the component model. In nLight, extra-functional properties are specified through an attribute type and their values by multiple attribute instances. For each value, metadata and the specification of the conditions under which the value is valid are enumerated. The purpose of this chapter is to highlight the key aspects of the implementation of the framework, namely:

- Introducing the mapping of the concepts of multi-valued context-aware extra-functional properties into a corresponding metamodel;
- Describing how a component model defined through a metamodel can be enriched with such properties;
- Characterising the extensible mechanisms for their specification.

[1] In this context, component model is not used in the strict CBSE sense. For example, UML or AADL could use nLight as they provide modelling elements that can be considered as components.

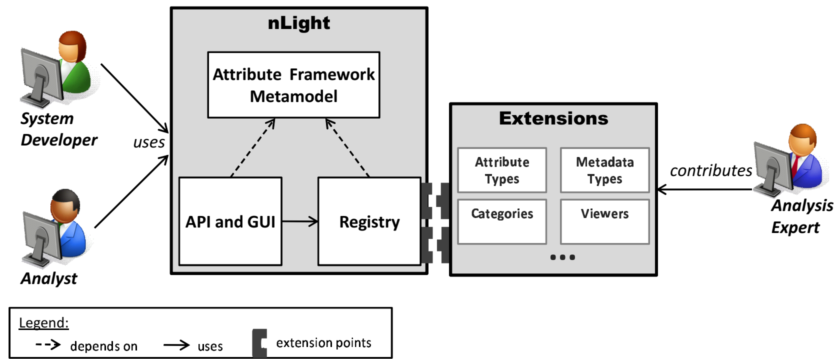


Figure 5.1: Overview of nLight's main constituents

5.1 Overview

The attribute framework, nLight, features two main parts complemented by a graphical user interface described in Section 5.5:

1) **The core part**
   This part provides the core functionalities offered by the framework. It is constructed around the "Attribute Framework Metamodel" based on which an API has been generated using the Eclipse Modelling Framework (EMF). This API provides basic methods to create, modify and delete the entities described in the metamodel. Depending upon this, a registry has been implemented to provides the necessary functionalities to register and handle attribute categories and types (attribute type or metadata type), and an API to support functionalities such as creation, modification of attribute instances.
2) **The extensible part**
   In order to support the high heterogeneity of extra-functional properties as described in Chapter 3, nLight must provide facility to be easily extended. This is done through the realization of extension points using the extension point mechanisms provided by Eclipse. These extension points provide, among others, the possibility to add new attribute types, metadata types, categories of extra-functional properties, and can be used as support for filtering mechanism and integrating new assessment techniques.

Figure 5.1 illustrates the relations between these parts with the core part corresponding to the Attribute Framework Metamodel, the API and GUI and the attribute registry, and the extensible part to the provided extension points and newly contributed attribute types, metadata types, categories and viewers for instance. In addition, Figure 5.1 also highlights the intended use of nLight. Analysis experts, who, for example, develop new analysis techniques, are in charge of contributing to the framework through the provision of suitable definitions of extra-functional properties. These definitions are then directly available to system developers and analysts.

5.2 Introducing Attributes

Following the definitions proposed in Chapter 3, multi-valued context-aware extra-functional properties are conceptually modelled through the *Attribute Framework metamodel*, which simplified representation is depicted in Figure 5.2. The dichotomy between attribute type and attribute instance is preserved through the relation between the *Attribute* and the *AttributeValue* metaclasses respectively. Also, corresponding to the attribute instance definition, an attribute value consists of data (*Data*), a set of metadata (*Metadata*), and possibly some validity conditions (*ValidityConditions*). The Attribute Framework metamodel is the cornerstone around which nLight is built.

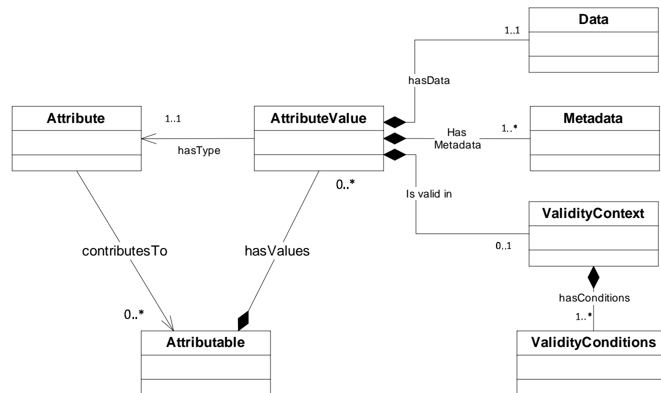


Figure 5.2: Simplified representation of the part of the Attribute Framework metamodel concerned with the multiple attribute instances.

5.3 Extending Component Models with Attributes

To have extra-functional properties attached to an element of a component-based design, its corresponding type must be *Attributable*, meaning that the Attributable metaclass is the entry point to nLight. In other words, the metaclass of the component model metamodel representing that element must extend the Attributable metaclass of the Attribute Framework metamodel as illustrated in Figure 5.3. For this simplified representation of a component model metamodel, components and interfaces are attributables. On the contrary, individual methods in interfaces cannot have extra-functional properties.

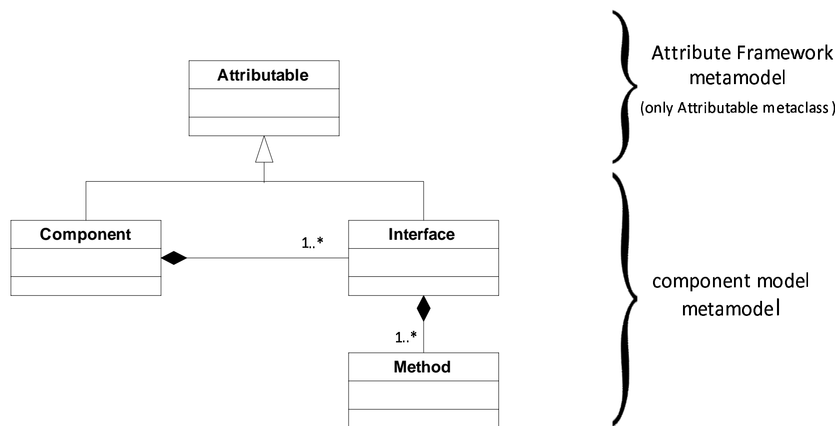


Figure 5.3: Defining Attributables for a Component Model

In order to enable the inheritance and refinement of extra-functional property values between component types and instances, we apply the inheritance refinement metamodel proposed in Chapter 4 to the Attribute Framework metamodel. As a consequence, two metaclasses have been added to the Attribute Framework metamodel as depicted in Figure 5.4: *ObjectType* and *ObjectInstance*. Both classes extend the Attributable metaclass and all model elements must inherit from one of them. Either the metaclass represents a type and, in this case, must inherit from ObjectType or it is an instance and in this case it must inherit from the ObjectInstance metaclass.

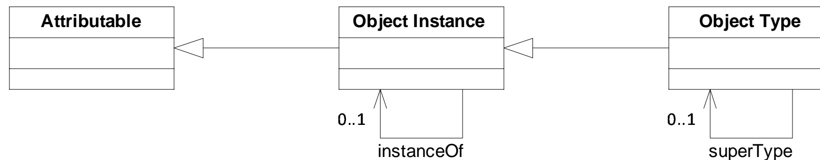


Figure 5.4: Attributable Related Metamodel (see Chapter 4.3.2 for details)

5.4 The Registry

The registry manages the specification of attribute types, metadata types and attribute categories. As shown in Table 5.1, the registry supports three types of functionality:

- Registration functionalities to store the specification of attribute types, metadata types, and attribute categories.
- Retrieval functionalities which main purpose is to enable getting lists of specifications contained in the registry. Specifications can be retrieved for a given type identifier, a category, or the registry can provide the complete list of all the specifications.
- Modification functionalities that help extending a specification with new items. Examples of this include adding a new predefined set of values for a given metadata, new contributions of attribute types to attributable, etc.

On the other hand, the registry prevents the suppression of types previously registered. This is done to ensure that previously used type specifications will always remain available.

The implementation of the registry is built around an API for the core functionalities mentioned above and uses the extension point mechanism provided by Eclipse to enable attribute type contributors to enhance the pool of existing attribute types with their own. To do so, two extension points have been defined: one to register new attribute and metadata types and one for new categories. Details on how to use these extension points to specify attribute types, metadata types and categories are explained in the following sub-sections.

Table 5.1: Overview of Functionalities Provided by the Registry

Type Specification	Registration	Retrieval	Modification	Unregistration
Attribute	Yes	All, Subsets	Addition of new items only (e.g. definition of new attributables)	No
Metadata	Yes	All, Subsets	Addition of new items only (e.g. definition of new predefined values)	No
Category	Yes	All, Subsets	No	No

5.4.1 Specifying Attribute Categories

Extra-functional properties can be classified in different categories as defined in various literatures such as [66] or [28]. However, not all classification sort the properties into the same categories and, what is more, categories are often named rather differently. In order to provide flexibility in the definition of the attribute categories, we have implemented a dedicated extension point. In contributing to this extension point as shown in the code 1, new categories such as the *"Timing category"* can be made available to sort attribute types. Only two parameters are required: *id*, a unique identifier for the category, and *name*, which is the corresponding user-readable name for the category.

Code 1: Specification of the timing category

```
< category id = "timingCategory"
            name = "Timing"
< /category >
```

In the attribute type specification, attribute can declare belonging to one of the contributed category. If no category is defined, a default category *"Misc"* is used.

5.4.2 Specifying Attribute Types

Following the attribute type specification introduced in Chapter 3, an attribute type is defined by a unique identifier, the list of attributables to which the property can be attached to, a suitable data format for the values, a list of supporting mechanism to manipulate the values of the instances and a documentation describing the extra-functional property and its usage. Accordingly, nLight is built around similar notions as detailed below.

Contributing Attributes to Selected Entities of a Component Model

In order to precisely define the entities of a component model to which a given extra-functional properties can be attached to, two pieces of information are required:

1) the metamodel of the component model that has been extended as explained in Section 5.3. This information is retrieved through the Uniform Resource Identifier (URI) associated with each metamodel.

2) the lists of metaclasses that are attributable for the property. This list is formed from the names of the corresponding metaclasses from the component model metamodel. If the list is empty, all the elements from the component model can be assigned with the given extra-functional property. On the other hand, if this list is not empty, then only the elements from this list can be assigned the given extra-functional property.

   For example, as shown in the extract of code 2, the extra-functional property *"worst-case execution time"* is attached to the component model specified with the metamodel "http://www.mdh.se/metamodel.ecore" and only ports, operations and components of this component model will be able to have extra-functional properties of this type attached to them.

The relation between attribute type (attribute), attribute instance (attributeValue) and attributable is depicted in Figure 5.5 which shows the part of the Attribute Framework metamodel concerns with these concepts.

Code 2: Contributing attribute to entities of a component model, an example

```
< attribute
  id = "se.mdh.progesside.attribute.wcet"
  name = "Worst Case Execution Time"
  dataPackageURI = ``http://DataTypeMM.ecore''
  dataType = "IntegerData"
  [...] >
  < targetPackage
    uri = ``http://www.mdh.se/metamodel.ecore'' >
    < ModelElementname = "Port"/ >
    < ModelElementname = "Operation"/ >
    < ModelElementname = "Component"/ >
  < /targetPackage >
< /attribute >
```

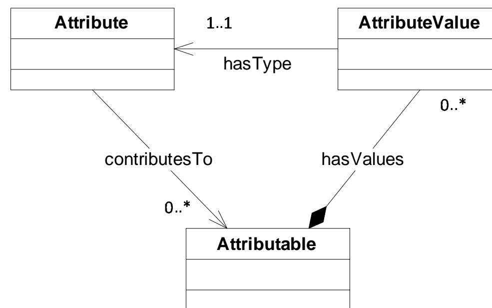


Figure 5.5: Attribute Type - Attribute instance metamodel

Identifier

As shown in Code 2 with the *"id"* attribute tag, each attribute type is specified through a unique identifier. To ensure the uniqueness of the identifier, we exploit Eclipse EMF and its capacity to generate universally unique identifier (UUID) conforming to the standard developed by the Open Software Foundation.

Defining the Attribute Data

As mentioned in Chapter 3, the data format must correspond to a data type used within a type system. For this, we have defined a generic and extendable data structure, represented by the abstract metaclass *Data* in the metamodel of the attribute framework as illustrated in Fig. 5.6. Data represents the value of an attribute instance. This metaclass can be specialized to create simple data types that can in turn be used to create more complex data types. Attribute type contributors can extend this structure with their own data type definition. Additionally, the data type definition must include operations on the type, such as a method to compare two data of a given type.

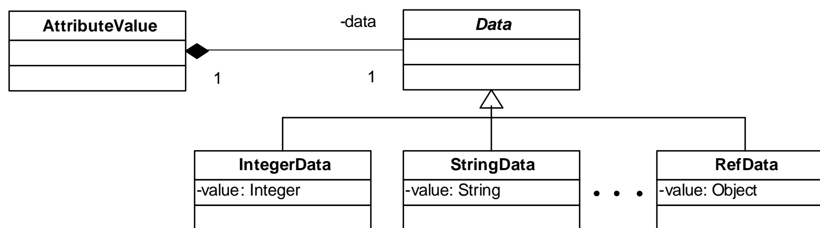


Figure 5.6: Attribute data.

Similarly to the specification of attribute types to the suitable entities of the component models, specifying the data type also requires two parameters: the metamodel of the data type to use and the name of the metaclass that represents the data format. Continuing with the *''Worst-Case Execution Time''* example introduced before in Code 2, this attribute uses IntegerData as format for the value with the dataType parameter. This format is defined in the `''http://DataTypeMM.ecore''` metamodel (dataPackageURI).

Configuring the usage

In addition to specify the data type for the attribute type, additional parameters must also be provided to configure its usage. These parameters relate to the supporting mechanisms described in Chapter 3 and are necessary to ensure that all the corresponding attribute instances will be manipulated in a uniform and consistent way. The following parameters are currently available in nLight:

Data Serialization The Data Serialization parameter specify how the data should be stored and retrieved.

Data Viewer This allows to specify how the value of the corresponding attribute instances should be visualised by using a dedicated viewer.

Data Editor Similarly, the data editor parameter indicates how the values should be modified.

Data Validator Data validators define constraints on the value. Such constraints include checking that the value is always positive, for instance. When the data validation fails, several actions can be taken:

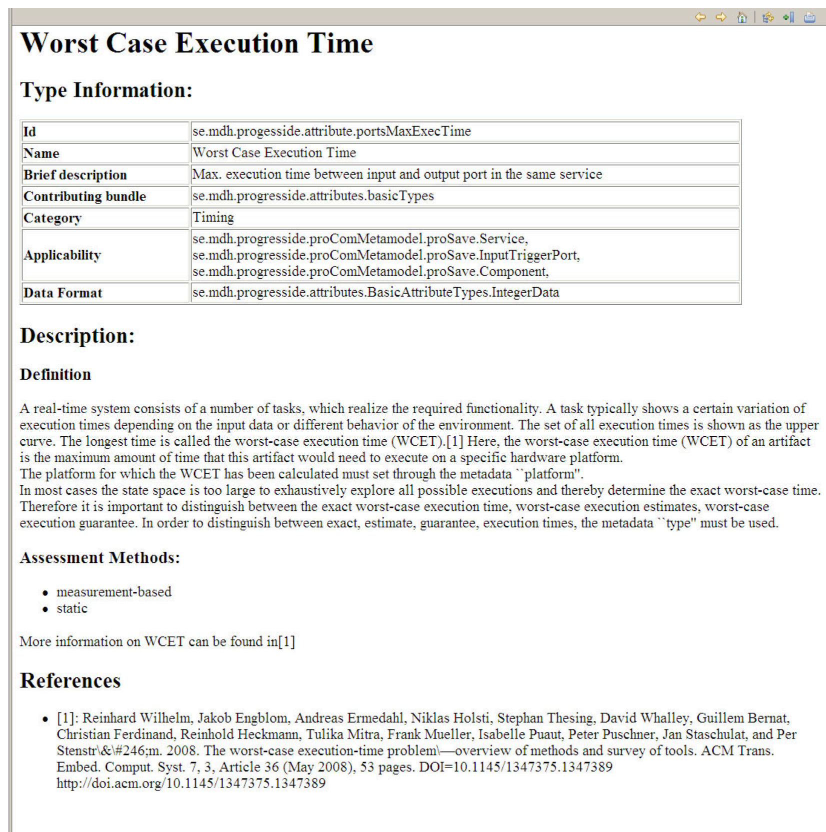
- *GUI_WARNING*: Raises an alert only. It is the responsibility of the users to fix it.
- *OPERATION_ABORT*: Prevent the creation of the value.

Inheritance Policies This parameter allows to define the inheritance policy, i.e, how the property value should be derived from a component type and to its instances in a controlled manner. The available inheritance policies have been defined in Chapter 4.3.2.

This set of parameters is not fixed. New parameters can be added to cover additional supporting mechanisms such as a parameter to support comparing and ordering attribute instances.

Documenting the property

Although not indispensable to the specification, it is important to properly document the extra-functional property being defined. This is necessary to explain to the intended users (system developers, analysts, etc.) how the property is expected to be used, since the users are not necessarily the contributor of the property. For that purpose, information regarding how its value must be represented, to which elements the property can be applied as well as supplying a precise definition must be provided. The first part of the documentation is automatically generated from the specification whereas the precise description of property (including its definition) must be provided as an HTML page. Figure 5.7 shows the final documentation rendered for the worst-case execution type property.



Worst Case Execution Time

Type Information:

Id	se.mdh.progesside.attribute.portsMaxExecTime
Name	Worst Case Execution Time
Brief description	Max. execution time between input and output port in the same service
Contributing bundle	se.mdh.progresside.attributes.basicTypes
Category	Timing
Applicability	se.mdh.progresside.proComMetamodel.proSave.Service, se.mdh.progresside.proComMetamodel.proSave.InputTriggerPort, se.mdh.progresside.proComMetamodel.proSave.Component,
Data Format	se.mdh.progresside.attributes.BasicAttributeTypes.IntegerData

Description:

Definition

A real-time system consists of a number of tasks, which realize the required functionality. A task typically shows a certain variation of execution times depending on the input data or different behavior of the environment. The set of all execution times is shown as the upper curve. The longest time is called the worst-case execution time (WCET).[1] Here, the worst-case execution time (WCET) of an artifact is the maximum amount of time that this artifact would need to execute on a specific hardware platform.
The platform for which the WCET has been calculated must set through the metadata ``platform''.
In most cases the state space is too large to exhaustively explore all possible executions and thereby determine the exact worst-case time. Therefore it is important to distinguish between the exact worst-case execution time, worst-case execution estimates, worst-case execution guarantee. In order to distinguish between exact, estimate, guarantee, execution times, the metadata ``type'' must be used.

Assessment Methods:

- measurement-based
- static

More information on WCET can be found in[1]

References

- [1]: Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenstr\&\#246;m. 2008. The worst-case execution-time problem\—overview of methods and survey of tools. ACM Trans. Embed. Comput. Syst. 7, 3, Article 36 (May 2008), 53 pages. DOI=10.1145/1347375.1347389 http://doi.acm.org/10.1145/1347375.1347389

Figure 5.7: Screenshot of the generated documentation for the WCET attribute type.

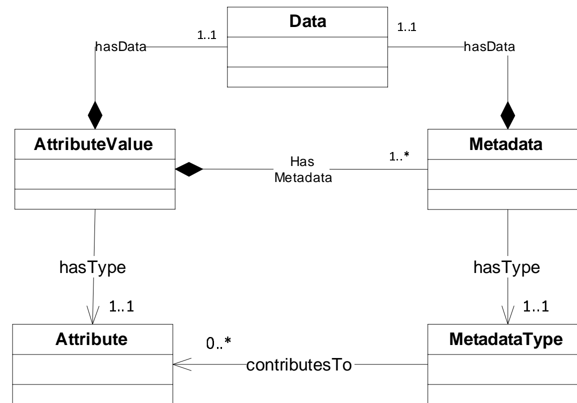


Figure 5.8: Simplified representation of the part of the Attribute Framework metamodel concerned with the metadata.

5.4.3 Specifying Metadata Types

Similarly to the attribute type, it is difficult to know beforehand the exact list of needed metadata. The use of metadata in the Attribute Framework is two-fold: *i*) it enables to distinguish between values and adds meaningful information to the way the value has been assessed and *ii*) it provides a support for automating the management of attributes in the framework. As a consequence, the proposed solutions to support metadata in nLight must be extensible. This is achieved through a similar solution as the one proposed for the definition of Attribute Type:

1) A dichotomy between Metadata Type and Metadata Instance (represented in Figure 5.8 by the metaclasses *MetadataType* and *Metadata* respectively), and

2) by the reuse of the extensible concept of data by which it is possible to specify the precise data format for a given metadata type.

In doing so, all the mechanisms defined for the data as described previously, can be made available for the metadata if needed.

Furthermore, as all metadata types are not necessarily meaningful for all attribute types, it is necessary to precise which metadata can be assigned to the values of a given attribute type. This is done through the relation contributesTo between metadataType and Attribute. For example, as explained in [65], worst-case execution time calculation methods fall into two categories (estimate or guarantee) and it is important to distinguish between them. A metadata *"TimingAnalysisType"* as specified in Code 3 can be used to express the nature of a worst-case execution time attribute for instance. Note, that the list of attribute types to which a metadata type can be attached to and the list of default values are not fixed at creation.

Code 3: Contributing a metadata type with predefined values to a subset of attribute types

```
< metadataType name = "TimingAnalysisType"
  ...
  < ContributesTo ContributesTo =
  "se.mdh.progesside.attribute.portsMaxExecTime"/ >
  < ContributesTo ContributesTo =
  "se.mdh.progesside.attribute.portsMinExecTime"/ >
  < DefaultValue DefaultValue = "estimate"/ >
  < DefaultValue DefaultValue = "guarantee"/ >
  ...
< /metadataType >
```

Cardinality

For some metadata, it makes no sense to have several metadata values of the same type to describe a value of an attribute instance. Such metadata types include for example version, modification time and accuracy. On the other hand, for others, several values might be used such as comment for instance.

The cardinality parameter of the metadata type definition enables to specify how many values of a given metadata type an attribute value can have. Three cases are proposed:

N attribute values can have at most N instances of this metadata type.
***** attribute values can have any number of instances of this metadata type.
\+ each attribute value must have at least one instance of this metadata type.

In addition, for each of the cardinality, the optionality of the metadata type must also be specified: whether the metadata type is *optional* or *mandatory*. A mandatory metadata type implies that it must always be provided whereas an optional one can be omitted.

Framework Metadata vs. Descriptive Metadata

Metadata are classified in two groups: the *framework metadata* and the *descriptive metadata*. The framework metadata are elements used to facilitate the implementation of the main functionality in nLight, and the implementation of more advanced mechanisms. Version, Creation Time and Modification Time are some examples from this group. These metadata are mandatory and must not be modified by the users of nLight, i.e. they are not be editable.

On the contrary, descriptive metadata are elements used to provide additional information concerning the attribute value such as the way the value has been obtained (Source metadata), or the platform on which that value is valid (Platform metadata). These metadata are optional and can generally be modified by the user of the framework.

Table 5.2: Characteristics of Framework and Descriptive Metadata

Metadata Type	Cardinality	Mandatory	Editable
Framework	1 (typically)	Yes	No
Descriptive	*	No	Yes

Having mandatory and non-editable metadata implies that those metadata are always defined by the framework. This requires the existence of default value setter that allows to set the value of the metadata upon creation. Any default value setter must implement the interface shown in the Interface Definition 4.

Interface 4: Standard interface for value setters

```
public interface DefaultDataValueSetter {
    /* This method sets the default value for the
       Data.                                        */
    /* Ret:  the newly created data                 */
    /* Arg1:  the data to modify                    */
    public Data setDefaultValue (Data data){
    ...
    }
}
```

5.5 The Graphical User Interface

The main entry point for a system developer or an analyst to use the nLight framework is the graphical user interface (GUI). As shown in Figure 5.9, the nLight GUI consists of two main parts:

1) the *Attribute List* which displays the list of attribute types available for a currently selected element of a component-based design. This list is sorted by attribute categories.
2) the *Properties Page* which displays the attribute instances currently attached to the selected element of a component-based design. This list is also sorted by attribute categories. For each attribute instance, its values with the corresponding metadata and validity conditions are displayed.

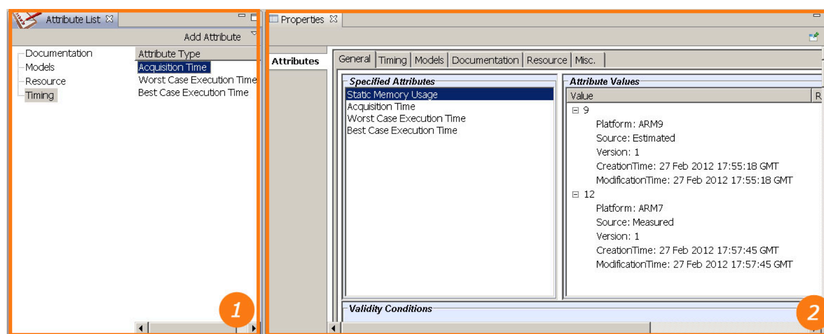


Figure 5.9: nLight's Graphical User Interface

5.6 Summary

In this chapter, we have described nLight, the framework specifically developed to support the seamless management of multi-valued context-aware extra-functional properties in component-based development. nLight aims at elevating extra-functional properties as first class entities in component models that do not per se provide support for them. As a result, nLight is not tight to a particular component model.

Through the repository and dedicated extension points, users can contribute to the framework in providing specifications of multi-valued context-aware extra-functional property as attribute types. The specification provides sufficient parameters to precisely define how the corresponding instances, i.e. the values, must be handled. Based on these specifications, the framework proposes a common graphical user interface in which users can *i*) see the complete list of properties available for an element of a component-based design, *ii*) visualise the documentation related to a given property, and *iii*) add, modify or remove in a uniform way extra-functional property values of an element of a component-based design.

Conforming to the definition of multi-valued context-aware extra-functional properties from Chapter 3, one of the key features of the framework is to allow several instances of a given attribute type to co-exist for an element of the component-based design. As a result, new values do not replace previously created ones. This, for example, allows to define early estimates even before the architectural element is implemented, hence enabling early reasoning on the design. Later, when the element is more mature, the early estimates values can be refined with the values obtained from analyses or, alternatively the early estimates can be kept as they are and new values can be created instead, thus allowing comparing between the different values. This is valuable when values have been assessed, for example, in using different analysis techniques. Furthermore, if nLight is closely integrated with different analysis techniques in a common development environment, values obtained from one analysis can served as inputs to other analyses.

Chapter 6

The ProCom Component Model

As pointed out in the introduction of the thesis, a key characteristic of embedded system development is the importance of producing reliable embedded systems in an efficient way. This is especially true for safety-critical and real-time systems. One of the foremost concerns to enable such a development is to satisfy the extra-functional properties. Others include the management of the functional complexity and the strong coupling with the hardware platforms. In that respect, the purpose of this chapter is to:

- Identify concepts and requirements suitable for a component-based approach for embedded systems development and its underlying component model.

- Define ProCom, the component model supporting the approach.

- Describe how the conjoint use of ProCom and nLight facilitates the assessment of extra-functional properties.

6.1 Domain Requirements for Component-Based Development of Embedded Systems

Embedded systems have specific requirements such as the timing demands and resource limitations, with corresponding development needs. In order to benefit from the known advantages of component-based development, one objective is to use components throughout the development process starting from a rough design of the system up to its final specification and deployable implementation. Another goal is to apply the component-based approach to the entire distributed system, not only within each physical node in isolation.

In our view, this requires the provision of a fully integrated approach managing traceability and dependencies between the artefacts generated during the development process such as source code files, models of entities, analysis results, design variants, etc. as well as providing means for various analysis techniques throughout the whole development process. Following this standpoint, a suitable component-based approach for distributed embedded systems should cover the whole development process starting from a vague specification of the system based on early requirements up to its final and precise specification and implementation ready to be synthesized and deployed. It should also be centered around a unified notion of components as a first-class entity gathering requirements, documentation, source code, various models, predicted and experimentally measured values, etc. and, improve the predictability of the developed systems by easily enabling various types of analysis, storing and managing the artefacts needed and/or produced by these analysis throughout the development process.

Combining these specific aspects together results in additional concerns for component-based development that are described in the following subsections.

6.1.1 Levels of Abstraction

The use of components throughout the whole development process means that the concept of a component spans a wide range between vague and incomplete specification to a very concrete one. During early design, components are used as very abstract entities. At this stage, the component just signifies a functional unit with no or very little detailed specification (see *(1)* in Figure 6.1), and the main objective is to decompose the system into smaller and more easily manageable units *(2)*. As the development process continues, the specification of components is refined and more details are added. Components also start having very concrete semantics such as specifying what happens when a

component sends a message *(3)*. Such a concrete specification is necessary to perform detailed analysis and to eventually map the system to a set of tasks and decide upon a deployment scheme. This process ends when the component or system satisfies the requirements *(4)*. At this point, components or the system can be synthesized with possibly the use of optimisation techniques.

However, in reality, this process from abstract to concrete is not necessary sequential:

- The exploration of different design possibilities implies the need to go back and forth between the abstraction levels.
- The level of abstraction of different parts of a system can differ significantly, due to the fact that critical parts are typically specified in more detail first in order to test the feasibility of the design. Moreover, reusing existing components, such as D from *(5)*, brings very well specified components into a system whose other parts may still be relatively abstract. This is visible in comparing component B and D in *(3)* in Figure 6.1.
- For some analysis techniques, it is necessary to keep a certain level of abstraction (or being able to abstract away from a detailed specification) as a more detailed specification would make the analysis method costly. At contrary, higher abstraction levels might not contain information necessary for a particular analysis and need to be concretized.

In summary, this means that components at different level of abstraction must be able to co-exist within the same model, and there has to be traceability between a component at a high level of abstraction and its concrete form.

6.1.2 Component Granularity

In distributed embedded systems, it is necessary to model the overall system structure, but also the detailed structure of individual parts, all the way down to the low-level control functionality. The granularity aspect naturally follows a decomposition pattern, in that it is possible to implement one component as a composition of smaller components.

However, the large span between the top and bottom of the granularity scale leads to components of very different size, and potentially of different semantics. Large components in such systems (e.g., the engine control in a car, or one production unit in an automation system) tend to be active (i.e., with their own threads of activity and possibly even including their own real-time scheduler), and encompassing complex functionality. Since the communication between

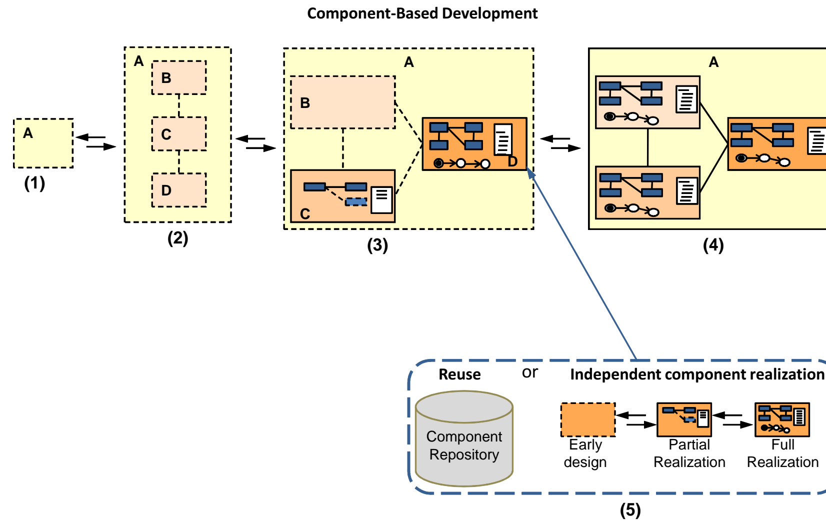


Figure 6.1: Co-existence of different abstraction levels.

these components often involves communication over a network (e.g., a CAN bus), it is typically realized by asynchronous messaging.

On the other side of the scale, there are smaller components responsible for a part of some control functionality, such as computing the deviation of a measured value from the desired value, or for communication with a single sensor or actuator. Since they represent composable low-level functional blocks, they typically do not posses their own threads. Also, the communication between them is much more tightly synchronized since most of the communication at this level is between components located on the same physical node.

6.1.3 Component vs. System Development

Component-based development distinguishes component development from the development of a system. This allows viewing components as reusable blocks which may be developed independently and at a certain point assembled to form a system. Although beneficial, this separation brings issues in the development of embedded systems, where the coupling between the hardware platform and the software is particularly tight. As a consequence, components can no longer be developed without some knowledge on the target platform where they are to be deployed. This is also true for many analysis techniques

(e.g., detailed execution time estimation, which is not possible unless the exact specification of the processor, memories, and compiler and linker flags are known). Therefore, it is important to find a suitable trade-off between platform-awareness and platform-independence. On one hand, information on the target platform on which the component is intended to be deployed or on which it has been used before, must be available. Yet, on the other hand, making the component platform dependent should be avoided as this restricts the possibilities to reuse it in different contexts.

6.1.4 Underlying Component Model

For a component-based approach to suitably support domain requirements, they must also be reflected by the underlying component model. This results in specific requirements being place on the component model. Directly derived from the domain requirements identified in the previous sections, the component model should:

- Cover the development process from early design up to the synthesis phase.
- Support the co-existence of different levels of abstractions and their interdependent relations.
- Simultaneously address the different requirements at different granularity levels.
- Make the component platform-aware while maintaining its development as platform independent.

Combining these requirements together leads to identify two orthogonal dimensions that must be supported by the component model. The first dimension is the abstraction level, which describes the successive refinement from abstract-to-concrete, i.e. from a rough sketch of a component to its final realisation consisting of source code, detailed timing and resource models for instance. The second dimension expresses the granularity level, i.e. the complexity and size of the components to realise. Figure 6.2 illustrates on an example the relations between these two dimensions with the abstraction dimension corresponding to the abstract-to-concrete scale, and the granularity level to the big-to-small scale. For example, an anti-lock braking system (ABS) that constantly adapts the brake pressure in accordance with the wheel speed to prevent wheel skidding while braking belongs to the big part of the scale. On the other hand, a brake force controller which task is only to monitor and adjust the pressure in a brake belongs to the small part of the scale. As illustrated in Figure 6.2, a component can be in different abstraction levels.

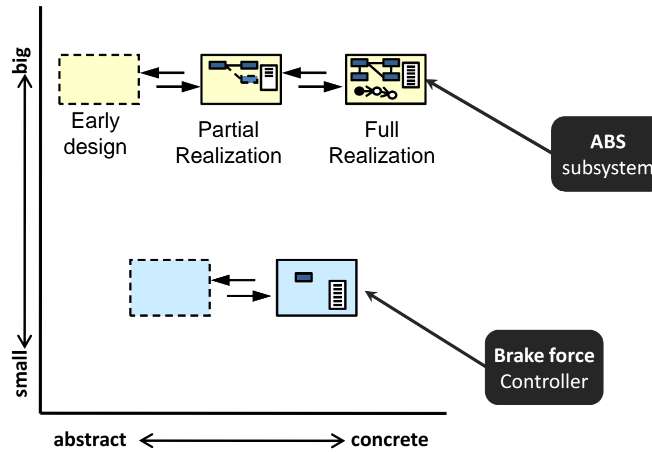


Figure 6.2: Two-dimensional scale

Ideally, a concrete realization should span the full range of all requirements. However, since the span of both granularity and abstraction levels is relatively large, it would result either in loosely defined concepts or in a very complicated component model to concretely develop. To mitigate complexity, the granularity and abstraction concerns can be divided into subsegments handled differently as illustrated in Figure 6.3.

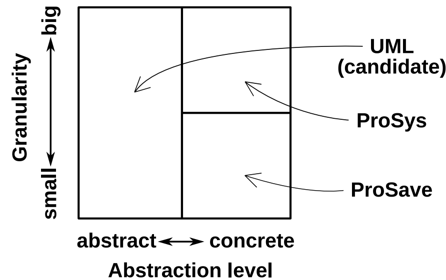


Figure 6.3: Partitioning example of the *granularity* and *abstraction* levels.

On the abstraction scale, the most abstract and concrete part are set apart from one another. In the most abstract part, the structure of the models has little in common with the concrete structure of the final system. For example, a system might be modelled there as a collection of use cases, by activity diagrams describing the overall system behaviour, or by entities representing different aspects of the system functionality and dependencies between them. These concerns are to a large extent covered by existing formalisms such as UML and UML profiles dedicated to embedded and real-time systems, such as SysML [70] or MARTE [71]. For most of these abstract modelling aspects, concepts related to "big" and "small" units as on the granularity scale can be found.

Note that each of the three segments in Figure 6.3 cover more than a single point on each of the two scales. The abstract part modelled in UML may range from very abstract use case modelling to relatively concrete specification of temporal requirements, etc. Similarly, the concrete side of the scale still covers several levels of abstractness, since the corresponding components at an early stage can be specified as black boxes[1], then gradually associated with more detailed models specifying its behaviour and internal structure, and finally given concrete source code implementation.

6.2 A Two-Layer Component Model

ProCom is a component model for distributed embedded system that has been developed as the concrete component model addressing the requirements described in the Section 6.1. The main characteristic of ProCom lays in layered-structure: ProCom consists of two layers, an upper-layer called *ProSys* and a lower-layer called *ProSave*. Components from both layers (ProSys and ProSave) are uniformly viewed as units of design and implementation that can be developed independently, stored in a repository, reused in multiple applications, etc.

[1] In this specific context, a black box component refer to a component for which the inner implementation has not been decided yet. It could be a primitive component, a composite component or a COTS or a legacy component.

ProSys covers the upper part of the granularity scale (i.e., the "big" units in Figure 6.3), and thus ProSys components are active and relatively independent. In ProSave, the lower layer, components correspond more to the constituents of the control functionality (i.e., the "small" units in Figure 6.3), and accordingly they are passive and more tightly coupled. As described later in this section, these two component types have different semantics and are also modelled in different ways. However, the two layers are not independent and cannot either be arbitrarily mixed. Instead, they are closely related since ProSave components can be used to constructs the internals of individual ProSys components (as described in Section 6.2.3).

In the rest of this section, we describe the two layers of ProCom and how they are related. The detailed description of the component model is available in [72] with its formal specification in [73].

6.2.1 ProSys — the Upper Layer

In ProSys, a system is constructed as a collection of communicating *subsystems*. Subsystems execute concurrently, and communicate by asynchronous messages sent and received through typed output and input *message ports*. This communication style is suitable at this level of granularity, since it allows transparent communication between subsystems independently of whether they reside on the same or different physical nodes.

Input and output message ports are not connected directly, but via *message channels* — explicit design entities representing data that are of interest to more than one subsystem. Multiple message ports (output- as well as input ports) can be connected to the same message channel, allowing n-to-n communication.

A benefit of these explicit message channels is that information about a message, such as precision, format and whether it should be available to diagnostic tools, can be associated with the message channel instead of with a port where the message is produced or consumed. This way, this information can remain in the design even if, for example, the producer is replaced by another subsystem. Also, since message channels can be introduced before any producer or receiver of the message has been defined, it permits early modelling of the run-time data managed by the system. Message channels also increase the awareness of the signal and information exchanged between subsystems.

ProSys allows hierarchical nesting of subsystems, i.e. a subsystem can internally consist of a collection of interconnected subsystems, accessible only through the ports of the enclosing subsystem. Contrasting such *composite* subsystems, a *primitive* subsystem is realized either directly by non-decomposable units of implementation (such as COTS or legacy subsystems), or by further decomposition in ProSave as described in Section 6.2.3.

Example

To illustrate ProSys, we use as an example the electronic stability control (ESC) subsystem of a car. This subsystem combines the functionality of the anti-lock braking (ABS) and traction control (TCS) systems, whose task is to prevent wheels from locking or spinning when braking or accelerating respectively, together with a stability control system (SCS) which handles sliding caused by under- or oversteering by reducing the acceleration and by braking individual wheels. In our example, braking is handled by the ESC subsystem itself, but to decrease acceleration it communicates with the engine (see Figure 6.4). Further, it reports its activity and dangerous conditions to the driver's information panel.

The internals of the ESC can also be modelled in ProSys, as shown in Figure 6.5. Inside, there are subsystems corresponding to specific parts of the ESC functionality (SCS, TCS and ABS). In our scenario, the TCS and ABS subsystems are reused from previous versions of the car, while SCS has been added to cope with under- and oversteering. These three subsystems compute responses based on their internal sensors and the speed of individual wheels, which is provided by a dedicated subsystem. The responses of the three subsystems are combined by the "Combiner" subsystem. The overall brakeage and throttle responses are forwarded to the "Brake valves" subsystem to regulate the braking pressure, and delegated to subsystems outside of the ESC, respectively.

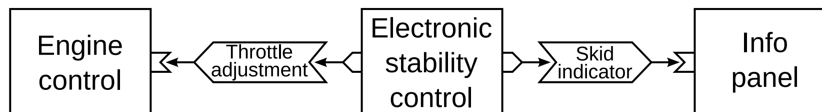


Figure 6.4: The connection of ESC to other subsystems. Message ports are connected via message channels.

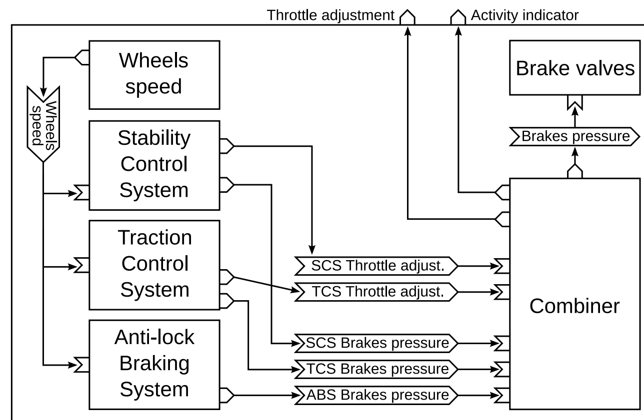


Figure 6.5: The ESC is a composite subsystem, internally modelled in ProSys.

6.2.2 ProSave — the Lower Layer

The ProSave layer targets the detailed design of subsystems allocated to a single physical node and interacting with the environment through sensors and actuators.

A subsystem can be constructed by a collection of hierarchically structured and interconnected ProSave *components*. These components are encapsulated and reusable design-time units of functionality, with clearly defined interfaces, but contrasting the subsystem "components" in ProSys, these components are closer in style to the *task* concept traditionally used when developing and analysing embedded systems.

ProSave is based on a pipe-and-filter architectural style with an explicit separation between data and control flow. The former is captured by *data ports* where data of a given type can be written or read, and the latter by *trigger ports* that control the activation of components. ProSave follows the push-model for data transfers and an input data port always contain the latest value written to it. Data ports always appear in a group together with a single trigger port, and ports in the same group are read and written together in a single atomic action.

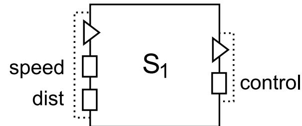


Figure 6.6: A simple ProSave component with one input group and one output group. Triangles and boxes denote trigger- and data ports, respectively.

Component semantics

ProSave components are *passive*, i.e. they do not contain their own threads of execution and thus cannot initiate activities on their own. Instead, each component remains in a passive state until one of its input trigger ports is activated.

In its simplest form, shown in Figure 6.6, a component has a single input trigger port, a single output trigger port, and a number of input- and output data ports grouped together with the two trigger ports. The semantics of such a component is that it is passively accepting data being written to the input ports until the input trigger port is activated. When this happens, the component switches into an active state, performing internal computations with the current value of the input data ports as input (and possibly based on the internal state of the component). The results of the computation appear atomically on the output data ports, together with an activation of the output trigger port. When the computation has finished, the component returns to the passive state.

More complex components can have several input port groups, each corresponding to a particular service provided by the component. Also, each service (i.e., each input port group) can have more than one output group, which allows parts of the result to be made available at different points in time, for example if some of the output is more time critical than the rest. Figure 6.7 shows an example.

The semantics of general ProSave components is not much different from the simple component semantics described above. The services are triggered individually, not the component as a whole. At a given point in time, each service is either in the passive or active state. When activated, a service only uses values from the input data ports of its own group. The internal state, however, can be shared between all services of a component. Similarly, a service can only produce output on the ports of its own output groups, and before the service returns to the inactive state again, each of its output groups must have been written once.

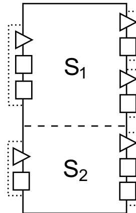


Figure 6.7: A ProSave component with two services; S_1 has two output groups and S_2 has a single output group.

Activations of the input trigger port of an active service are ignored, but input data ports can receive data while the component is active. This data, however, can not affect the current computation, but will be used as input the next time the service is activated (unless overwritten by a new value before that). This means that once a service has been activated it is functionally (although not temporally) independent from other components executing concurrently, which simplifies analysis.

Primitive and composite components

ProSave components come in two basic types: *primitive components* realized by code, and *composite components* realized by a collection of subcomponents. For a primitive component, each service is implemented by a non-suspending C function. There is also one function called at system startup to initialise the internal state of the component. Figure 6.8 shows an example of the header file of a primitive component.

Composite components internally consist of *component instances*, *connections* and *connectors*. A connection is a directed edge which connects two ports of compatible types. Connections go from an output port to an input port, but in this respect, the ports of the enclosing composite component are inverted (meaning that, for example, an input port of the composite component can be connected to an input port of one of the components instances inside). Connectors, on the other hand, are constructs that provide detailed control over the data- and control-flow inside a composite component. The connectors in ProSave are selected to support typical collaboration patterns, but the set of connectors is expected to grow over time as additional data- and control-flow constructs prove to be needed. The initial set includes connectors for *forking*

```
[...]
typedef struct save_S1_cpt save_S1_cpt;
typedef struct save_S1_S1_svc save_S1_S1_svc;

typedef enum{
        SAVE_S1_S1_STARTING,
        SAVE_S1_S1_triggerOut,
        SAVE_S1_S1_FINISHED
} SAVE_S1_S1_COMPUTATION_STATE;

struct save_S1_S1_svc
{
        save_S1_cpt *cpt;
        char activated;
        int current_state;
        SAVE_S1_S1_COMPUTATION_STATE computation_states[2];
        int triggered_outputs;

        int *in_speed;
        float *in_dist;
        int *out_control;
        char control_updated;
};

[...]

struct S1_state
{
  // Start of user code state variables definition
  [...]
  // End of user code
};

void entry_S1_S1(save_S1_S1_svc * svc);
void S1_init(save_S1_cpt * cpt);
```

Figure 6.8: Excerpt of the header file of the component in Figure 6.6

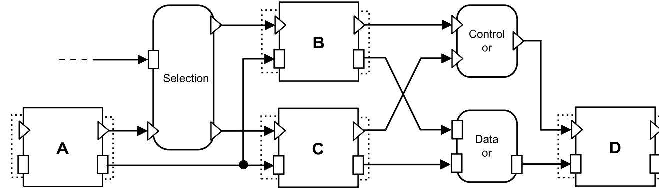


Figure 6.9: A typical usage of *selection* and *or* connectors. When component A is finished, either B or C is executed, depending on the value at the selection data port. In either case, component D is executed afterwards, with the data produced by B or C as input.

and *joining* data or trigger connections, and for dynamically *selecting* a control flow path depending on a condition. Figure 6.9 shows a typical usage of the *selection* connector together with *or* connectors. For a complete description of all connectors, see [72].

6.2.3 Integrating the Layers: Combining ProSave and ProSys

The integration of the two ProCom layers allows a primitive ProSys subsystem to be further specified using ProSave. Concretely, this is done similarly to how composite ProSave components are defined internally — as a collection of interconnected components and connectors — but with the addition of clock to specify periodic activation of ProSave components. A clock has a single output trigger port which is repeatedly activated at a given rate, its period.

To achieve the mapping from message passing to trigger and data, and vice versa, the message ports of the enclosing primitive subsystem are treated as connectors with one trigger port and one data port, when seen from inside the subsystem. An input message port corresponds to a connector with output ports, and whenever a message is received by the message port, the message data is written to the data port and the trigger port is activated. Oppositely, an output message port corresponds to a connector with an input trigger and input data ports. When triggered, the current value of the data port is sent as a message.

In addition to strictly periodic activation, ProCom also supports aperiodic activation such as events initiated by external devices. Aperiodic activation are handled locally by each component responsible of an external device and are modelled through an event connector.

Example

Modelling in ProSave and its connection to ProSys is illustrated on the SCS subsystem from the previous example. The SCS acts as a primitive subsystem on the ProSys level, meaning that it is not elaborated in ProSys any further. It may be realized either directly by code or by elaborating it in ProSave (thus changing the level of granularity). We have chosen the latter — see Figure 6.10. The SCS consists of one periodic activity, which runs at a frequency of 50Hz (specified by the clock). When activated, it first reads the data from sensors. Based on their outputs and the speed of individual wheels (obtained from the latest “Wheels speed” message) it computes the actual direction of the vehicle and the desired direction indicated by the steering wheel. After both computation components have finished, the “Slide detection” component compares their results (i.e., the actual and desired direction) and determines whether any action is required to ensure the stability of the car. The last component in the chain computes the actual response of the SCS, which consists of adjustments of brakeage and acceleration.

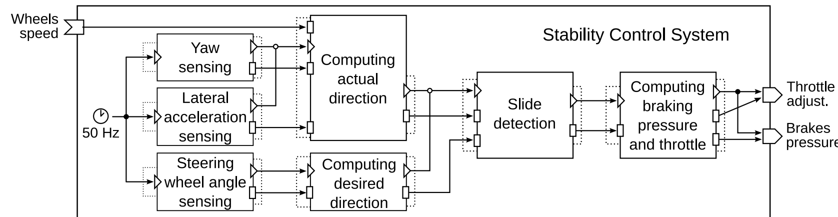


Figure 6.10: The SCS subsystem, modelled in ProSave. The dots and circles are shorthand notation for *fork* and *join* connectors, respectively.

6.3 Extra-Functional Properties in ProCom

ProCom has been developed to facilitate the expression and analysis of functional and extra-functional properties through *i*) its layered-structure and in particular the restrictive execution semantics of the lower layer, and *ii*) its concepts of rich design-time components. However, contrary to other component models and approaches such as PECT [16], or Fractal [43] that provide dedicated extra interfaces for managing properties, ProCom does not, per se, provide such capability.

Instead, through its concept of rich design-time component, ProCom is at a junction between component-based development and model-driven development. From the component-based development side, ProCom builds upon structuring the system out of well-defined pieces of functionality that can be independently developed, analysed and reused. From the model-driven development side, ProCom acknowledges the need that different models are used for different purpose throughout the development process. In order to meet aspects of component-based development, the scope of these models is limited to individual components for which they provide dedicated views on additional concerns. This allows to package any artefacts required or produced during the development of a component together, hence providing suitable ground for different analysis at different phases of the development process. Note, that it is important that the models are kept consistent during the development process.

For example, as illustrated on Figure 6.11, a ProSave component A can be directly analysed using the architectural model only. This includes analysis such as checking compatibility between ports, assessing timing properties, etc. With new models available, additional types of analysis can also be supported as shown in Figure 6.12. An example of such analysis is provided by the integration of REMES [74], a formalism for the design and analysis of resource-constraint component-based embedded systems that enables model-checking to verify functional, timing and resources-related properties (see Chapter 8). Similarly to what has been done in [75] with UPPAAL PORT and SaveCCM, analyses can be improved by the combined usage of both their specific model and the architectural model.

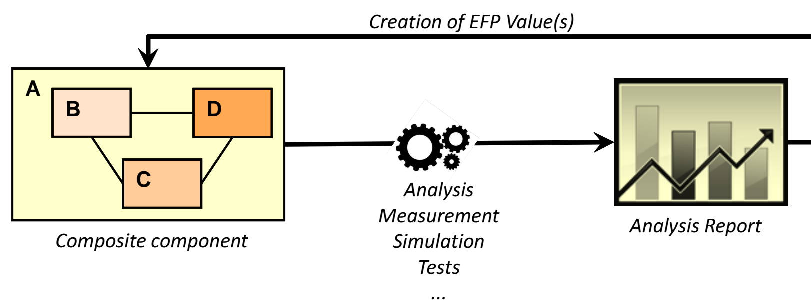


Figure 6.11: Analysis directly derived from a ProCom design.

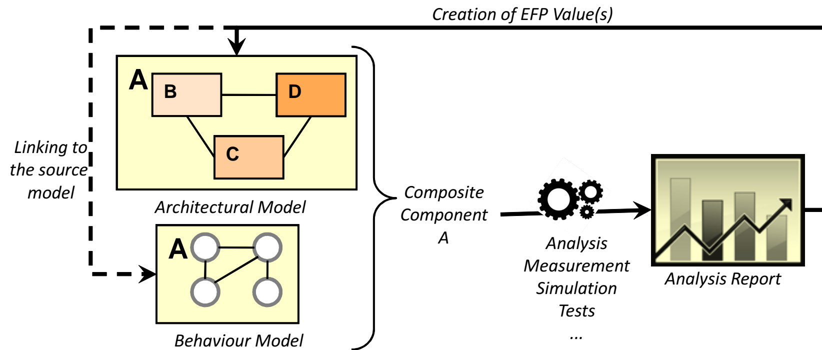


Figure 6.12: Analysis based on dedicated models.

6.4 Summary

In this chapter, we have started by investigating which requirements must be met by a component-based development approach to suitably support the development of distributed embedded system. In particular, one of the characteristics of this approach is to flexibly cover the whole development process from early design up to synthesis while, at the same time, facilitating various analysis to be performed at any step of the process. Based on the identified domain requirements, we have then developed ProCom, a component model for the development of component-based software embedded systems.

Through its hierarchy of interrelated layers, ProCom enables to address the different concerns that exist at different levels of granularity within a single formalism to build distributed embedded systems. Furthermore through its concept of rich-design time components, ProCom allows to gather any development artefacts as components, hence placing them as predominant entities of the development and facilitating their reuse. Finally, ProCom facilitates analysis of certain properties in early phases of the development process thanks to its formally specified semantics.

Chapter 7

PRIDE: The ProCom Integrated Development Environment

In the context of the thesis, the evaluation of the approach of merging component-based principles and embedded system development needs requires the implementation of a complete development toolchain that *i*) covers the necessary activities from component-based design up to synthesis and deployment, and *ii*) supports and integrates various analysis techniques throughout the development process. In the previous chapters, we have described the contributions of the thesis following two main lines of work in component-based development, namely the management of extra-functional properties and the specific requirements for embedded system development. In this chapter, we will show how these contributions have served as a basis to build integrated development environments supporting them. This chapter starts by describing lessons learned from the development and use of an initial prototype built for SaveCCT [14], a component-based development approach for embedded systems that enables early formal analysis of timing properties. Then, it presents how these lessons having been taken into consideration in the creation of the PRIDE tool suite.

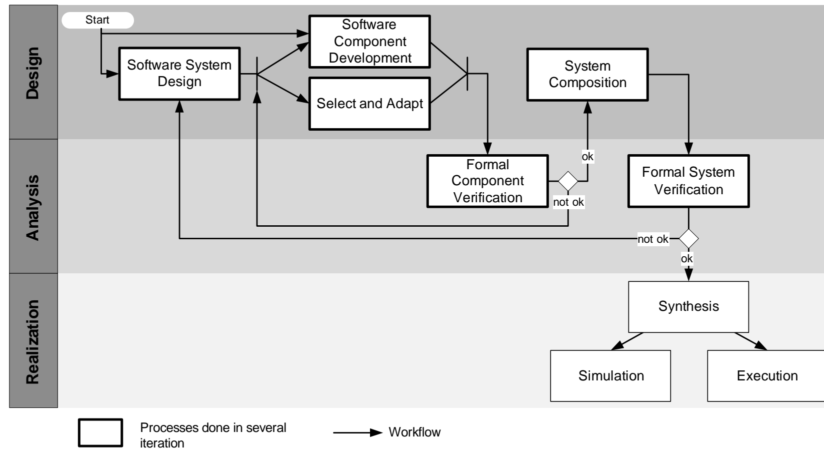


Figure 7.1: Overview of the SaveCCT development process

7.1 Feedbacks from an Initial Prototype

Our work on integrated development environment started with building an initial prototype based on the concepts, methods and techniques previously developed for SaveCCT [14]. The approach also aims at component-based development for dependable embedded systems. It represents a simple use-case scenario of the approach described in Chapter 6 in which the use of components is restricted to the design only, the analysis is performed on system-scale and the synthesis is a single-step activity performed at the end of the development process.

Accordingly, the *Save Integrated Development Environment* (Save-IDE) has been specified and developed to support the requirements and constraints of the SaveCCT approach together with SaveCCM, its underlying component model [76]. In addition, all the tools have been integrated in strictly following the exchange format specified in the SaveCCM reference manual [76]. The remainder of this section starts by giving an overview of the SaveCCT development process in Section 7.1.1 before describing the IDE itself in Section 7.1.2.

7.1.1 Intended Software Development Process

In SaveCCT, the development process is designed as a top-down approach with an emphasis on reusability. It includes three major phases: Design, Analysis and Realization, as illustrated on Figure 7.1.

The process begins with the *system design* phase in which the system is broken down into subsystems and components compliant with the SaveCCM Component Model [77]. Following this decomposition, system requirements are transformed into component requirements used as a basis to determine the next step of the development process. If already existing components (partially) matching the requirements exist, the *select and adapt* activity is taken. Otherwise, new component(s) need to be developed (i.e. the *component development* activity is taken).

Correspondingly, the components are first analyzed and verified individually towards the requirements (*formal component verification*). In a following phase, after having reconstructed the system (or parts of the system) out of individual components and their assemblies (*system composition*), the obtained compositions also need to be analyzed and verified (*formal system verification*). As long as the results produced in those analysis steps do not satisfy the requirements, i.e. some problems in the design still exist, the design of the system is supposed to be modified and checked again against the requirements.

When the results are acceptable from an analysis point of view, the *realization* phase starts. It consists of *synthesis* activity in which the system is synthesized automatically based on the input from the system design, on the implementations of the components and, on static algorithms for the resource usage and timing constraints. All the necessary glue code for the run-time system is produced. The resulted image can then be tested on a simulator or downloaded into the target platform.

To reduce the risks of errors in manual activities, and to increase the development efficiency, several parts of this process are automated. A first automated activity is the production of the skeleton of the implementation files (C files and their corresponding header files) based on the specification of the component. Another one is the generation of the interchange file used as communication medium between tools. This interchange file transforms the system design into an XML-based representation as specified in [77]. The third one occurs during the synthesis which includes transformation of components into the executable real-time units, tasks, glue code generation, inclusion of a particular scheduling algorithm, compilation and linking all elements in the executable image.

7.1.2 SaveIDE — the Save Integrated Development Environment

Save-IDE[1] is designed as a platform that facilitates the integrations of tools compliant with the exchange format specified in [76]. It is developed as a set of plugins for the Eclipse framework. As illustrated in Figure 7.2, it supports three key activities of the development process: *i*) *component-based design* that distinguishes between system and component development and includes modelling and design of the components, the architectural design of the system and specification and implementation of components, *ii*) *analysis* of the system and the components, and *iii*) *synthesis* that includes transformation from components to tasks, setting up execution parameters like priorities and periodicity of execution, glue code generation and compilation.

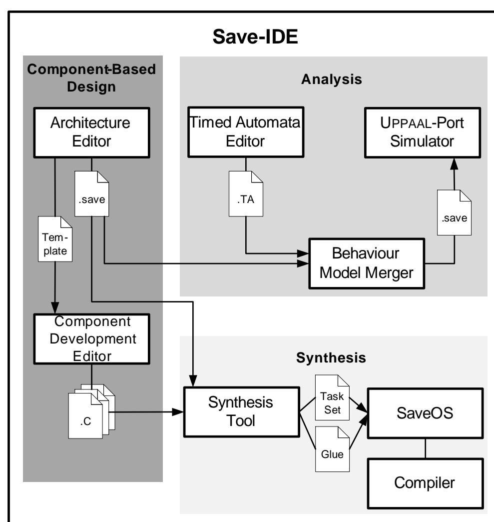


Figure 7.2: Overview of the Save-IDE tool-chain

[1]The Save-IDE is available for download from the web page http://sourceforge.net/projects/save-ide/

Figure 7.2 also shows the tools involved in each of these activities and how they are organised together. A screenshot of the environment is provided in Figure 7.3. Eight tools are currently integrated into the Save-IDE: the *Architecture Editor*, the *Component Development Editor* for the component-based design; the *Timed Automata Editor*, UPPAAL port with its simulator and formal verifier based on UPPAAL for the analysis; and the *synthesis tool* targeting the *SaveOS* and a *compiler* for the synthesis activity. The remainder of this section describes these tools per activity.

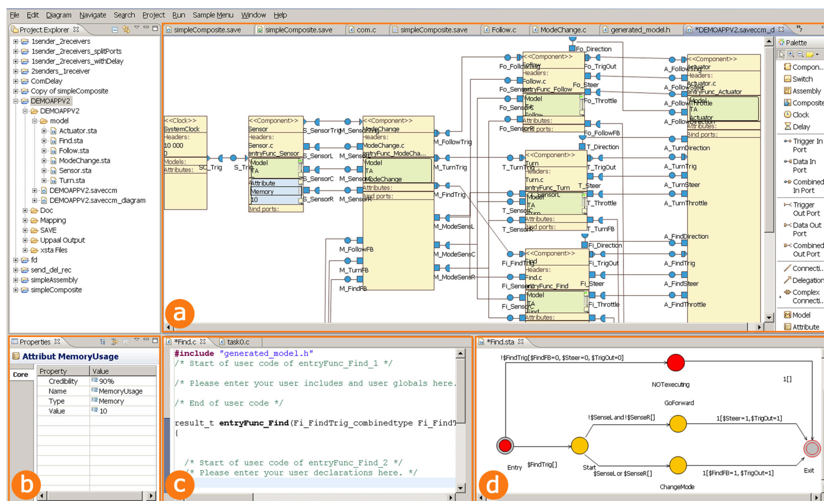


Figure 7.3: Screenshot of the Save-IDE

Component-Based Design Toolset

In SaveCCT, the design of a system distinguishes between two independent activities: *software system design* and *software component development*. Software system design consists of designing a system out of independent and possibly already implemented components, i.e. components being produced through the component development activity. Alike, the *Architecture Editor* (see *(a)* in Figure 7.3) supports both.

It enables creating systems and components compliant with the SaveCCM component model. To do so, the tool provides support for designing systems and components with the set of architectural elements prescribed by SaveCCM. These elements are component, assembly, composite, clock, delay and switch. Furthermore, the component model also enforces the "pipe-and-filter" communication paradigm distinguishing between control-flows with trigger ports and data-flows with data ports when the architectural elements are connected together.

For each composite architectural element (e.g. assembly, composite and switch), two views coexist: the *external view* (see *(a)* in Figure 7.4) and the *internal view* (see *(b)* in Figure 7.4). The external view describes the name and type of the element, the ports, and the models annotated to the element (such as time behaviour represented by a timed automata). In other words, the external view specifies the component interface. On the other hand, the internal view handles the inner elements and their connections only. This view can be hierarchical since SaveCCM allows hierarchical compositions of components and assemblies. This separation is done through partitioning of diagrams which allows having a clear distinction between the design of the external from the design of the internal elements. For SaveCCM "primitive" components, only the external view is available. Their internal view corresponds to their implementation within the component development editor.

In addition to the specification of functional interface, the Architecture Editor makes it possible to assign different attributes to the components, such as execution time, or behavioural model as visible in the properties page *(e)* in Figure 7.3.

The Component Development Editor (see *(c)* in Figure 7.4) is realized by the integration of the Eclipse C/C++ Development Tooling (CDT) that provides the features required for the implementation of the primitive components in C language. To increase development efficiency and reduce the risks of errors in manually translating the component interfaces into code, skeletons for the implementation files are generated directly from the specification of the component. The skeletons for the C and header files contain the mappings from ports to variables and function headers. As a result, the component developer only needs to implement the component functionality.

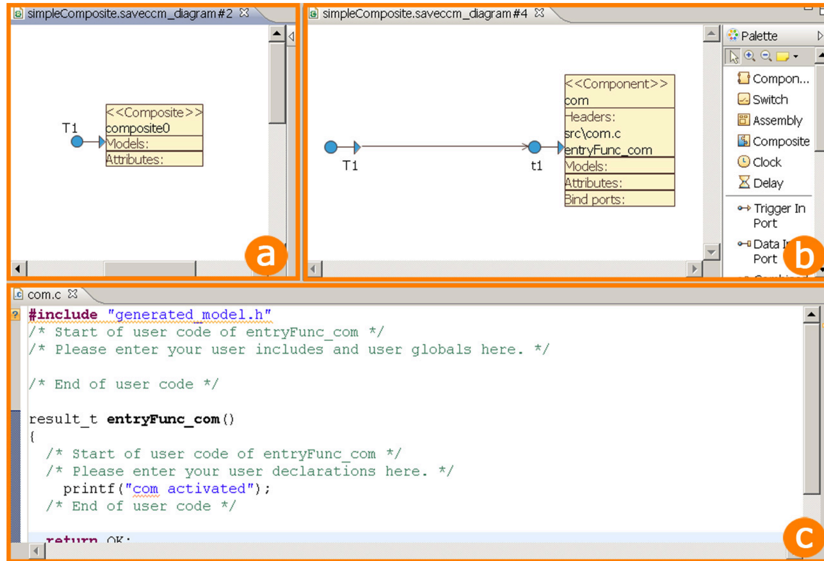


Figure 7.4: Screenshot of the tools involved in the component-based design, in which *(a)* shows the external view of a composite component, *b)* its internal view and *(c)* the component development editor for the “com” primitive component.

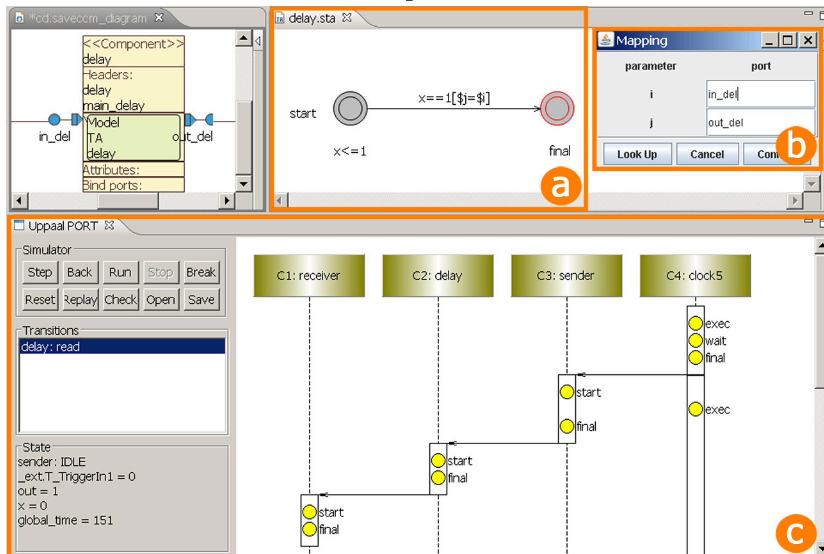


Figure 7.5: Screenshot of the tools involved in the analysis, in which *(a)* shows the timed-automata editor for a delay component, *b)* the tool facilitating the mapping and *(c)* UPPAAL PORT.

Analysis Toolset

The analysis part in the Save-IDE supports the theoretical contributions on partial order reduction techniques proposed by Håkansson and Pettersson in [75]. These techniques exploit the specific execution semantics of SaveCCM components, restricted to a "read-execute-write" execution sequence, and the hierarchical structure of the system. The main purpose of this semantics is to be able to perform model-checking analysis of the system in the early phases of the development process without requiring any component implementation. To support these techniques, several tools have been implemented and/or integrated in the Save-IDE: a Timed Automata Editor, a simulator, and a model-checker.

The Timed Automata Editor (TAE) (see *(a)* in Figure 7.5) provides developers with a graphical user interface for creating formal functional behaviour and timing models of SaveCCM components. The models are expressed in a timed-automaton formalism and can be created independently of the targeted component. This increases the reusability of the model but requires means to associate it with a corresponding component. In the Save-IDE, this association is done in a semi-automatic mapping process. First, it requires the user to manually create a "TA" attribute in the architectural editor that points to the location of timed automata model. Then, the external ports of a SaveCCM element are mapped with the variables of the corresponding timed automata (see *(b)* in Figure 7.5) .

Once every component in the system have a timed-automata model, the system can be analysed using UPPAAL PORT (see *(c)* in Figure 7.5) . UPPAAL PORT is implemented as an extension on the UPPAAL model-checker [78], and features a graphical simulator and a formal verifier. UPPAAL PORT requires a specific XML-format that is automatically generated from the Save-IDE by merging together the architectural description of the system compliant with the SaveCCM exchange format, the output of the TAEs and the mapping files. Using the simulator, it is possible to explore the dynamic behaviour of a complete SaveCCM design. In this way, designers can validate the design and gain increased confidence in the design. Using the verification interface, it is possible to establish by model-checking whether a SaveCCM model satisfies formal requirements specified as formulas in a subset of the logic Timed CTL. This helps to further increase confidence in the component-based design, w.r.t., e.g., functionality and timing. More information on UPPAAL PORT can be found in [79].

Synthesis Toolset

SaveCCM systems can be automatically synthesized using the synthesis tools integrated into the Save-IDE. The synthesis enables to transform the component-based design of the system into an execution model that can then be compiled before being installed on the target platform. External tools, such as CC-Simtech [80], can also be used to simulate the system on a standard desktop computer.

The synthesis takes the SaveCCM model and constructs a set of trees based on the applications triggers. These trees are then used to generate the software code realized into the tasks, i.e., the function calls to the software components as well as glue code needed for passing data between the components. Each tree is mapped to one real-time task, and the configuration of the task is done with respect to the parameters of the trigger, e.g., setting of periods and priorities. The synthesis is performed towards the Save Operating System (SaveOS), which is an abstraction layer allowing systems to be ported to different operating systems and hardware platforms. SaveOS is designed and implemented in a way that it requires minimal computing and memory resources and provides a neglecting overhead. It enables systems to indirectly call native operating system services through the SaveOS application programming interface. Likewise, the configuration of the run-time environment can be changed without having to change the system design or the implemented behaviour of the components.

7.1.3 Lessons Learned

This environment has been used internally by the members involved in its realisation and externally by students outside the projects to develop diverse small applications. In [81], a comparison between Save-IDE and a professional tool enhanced with a profile for SaveCCM has been performed. This experiment is performed on a small group of students concerns only the modelling aspect of the environment. Yet the students' feedback show some indications that a dedicated design environment is more efficient than a general-purpose environment customized to fit a particular need.

The environment has also been used in [82] and in [24], in which an industrial control system and a simple truck application have been realized respectively. Those two examples show the feasibility of the integrated approach. In particular, they highlight the possibilities of tightly interconnecting design and formal analysis tools, which enable formal analysis of on-going design already in an early design phase.

Several lessons have been drawn from the development and use of this integrated development environment. These are the following:

1) **Component as a central and uniform development unit** Despite its precise specification, the concept of component in the Save-IDE is ambiguous. One of the cause of this ambiguity stems from the presence of other concepts such as assembly, composite, system. These concepts are only distinguished from one another by the execution semantics which is restricted for composite and primitive components. At design-time, this principal difference is not intuitive for the users. Moreover, components are considered as one of the artefacts used during the development process. Other main development artefacts include timed-automata models, source codes, etc.

   Due to that, the various artefacts used or produced during the development process are not tightly bounded to their corresponding components. For instance, analysis models such as timed-automata models, do not necessary belong to the file structure of the components as shown in the project explorer in Figure 7.3. This implies that upon reuse of a component, the models that have been specified as attributes must be retrieved and place in the exact location specified by the attribute in the new environment. This is a cumbersome process that limits the reusability of the analysis models of a component and of the component itself.

   Accordingly, components must instead be considered as the main development units. In that view, a component should be seen as a placeholder (somewhat similar to the concept of package in object-oriented programming): it enables gathering the different artefacts corresponding to the component. As a result, a component should be then the collection of assets created or required during the development process. These assets correspond, for example, to architectural models, behavioural models, source code, tests, documentation, etc., and must be kept consistent. This can be seen as integrating aspects of model-based development into component-based development.

2) **Enforcing component type and instance** In the Save-IDE, components are essentially design entities that are directly created within the design of a composite entity such as system, assembly or composite composite. The benefit of such an approach is to provide a lot of flexibility in

the design. However, this leads to unconsciously intermixing component types and component instances. In taking the available possibility of copying components directly within a design of a composite element, this leads for example to have two instances of a component type. Users are then able to modify one of the instances, which implies either *i*) they are instances of two different component types, or *ii*) they are instances of the same component type and therefore the second instance should be modified too to keep the consistency between the two instances. In other words, this means that an instance of the component can be modified independently of its component type and consequently, ensuring consistencies of a component type with its instances and implementation requires numerous checking. As a consequence, one problem with this approach is that it is difficult to determine when the design of the component is completed and must not be changed any longer.

3) **Flexible and multi-step synthesis** In the current approach supported by Save-IDE, the transformation of the design model into an execution model allowing synthesis and optimisation steps is performed at the end of the process only, after the design has been verified and validated. Yet, the validation and verification are performed at a high-level of abstraction without connection to the component implementation used in the synthesis and without any specific information regarding the target platform. It is assumed that the implementation does not break the behaviour formally modelled. This can have some negative effects on the efficiency of the approach when the fully implemented system does not meet its timing requirements or the timing requirements are not feasible. The development process might then start over at the design step with the re-design and re-implementations of the erroneous parts. As a consequence, the validation and verification steps must be carried out again. Furthermore some analysis techniques, such as schedulability, cannot be performed on a high-level of abstraction. Some potential solutions that need to be further investigated are to connect implementation with analysis or generating implementation from the models used by the analysis techniques. Also, synthesis must be viewed as more complex than a single-step operation performed at the end of the development process. It requires many analysis, tests and optimisations that are closely related to the design, implementation and various extra-functional properties such as timing or resource usage, and must therefore be also tightly connected with them.

4) **Lifting the importance of extra-functional properties** In Save-IDE, attributes are viewed as simple means to display information on extra-functional properties extracted from analysis results or alternatively as a convenient support to link an analysis model with a component. Yet, most of the time, the attribute elements were rarely used at all. This is due to the lack of a clear purpose for the attribute concept in the SaveCCT approach together with an unclear specification of the concept.

7.2 Concepts behind PRIDE

Since the theories underlying the creation of the Save-IDE tally with several aspects of the contributions of the thesis, our initial intention was to reuse the Save-IDE and modify it to support the novel ideas. As for the Save-IDE, the aim is to support component-based software development of embedded systems in a process spanning from early specification up to synthesis. However, instead of mainly considering early formal analysis, it is envisioned that an interlacing of analysis techniques is to be applied at different stages of the development process. The development process is intended to be flexible and to enable suitable information from one of the development activities to be available in the others. For example, information on the target platform on which a component is planned to be allocated to should be available to the analysis. Likewise, results from analysis should be available in the software design. After balancing these aspects with the features provided by the Save-IDE, we decided to develop PRIDE, the ProCom Integrated Development Environment, as a new integrated development environment.

The knowledge and experience gained from the development and use of the Save-IDE have been integrated into PRIDE. PRIDE is designed as a standalone environment that can be easily extended through, for example, the integration of new analysis techniques. PRIDE is centered around the notion of components as main development artefact. A component is considered as a rich design-time concept that corresponds to the collection of all the related development artefacts that are needed, specified and produced during the development process. In other words, in addition of having clear functional boundaries derived from its component model, a component consists of requirements, documentation, source code, various models (e.g. behavioural and timing), predicted and experimentally measured values (e.g. performance and memory consumption), etc.

PRIDE also allows components of different maturity, from early specifications to fully implemented components with more detailed information, to co-exist within the same model and to be manipulated in a uniform way. This provides ability to leave component realization undecided. The component realization decision can be thus postponed while still being able to reason about the design of the system. For example, this allows to perform different analyses in early stages of development process based on the software architecture and provide system architects with early estimates on system behaviour and properties. In this way, possible problems can for example be detected before the system is implemented and avoid late changes.

Reusability is also one of the key concepts in PRIDE, aiming to significantly shorten development time. The tool introduces the distinction between component type and component instance. Each use of a component type creates a component instance of the given type, and by editing a component type, all its instances are affected. To foster reusability, components can be stored in (and imported from) a shared repository, making them available for reuse in different projects. As a result of the rich design-time component concept, component reuse implies reuse of component properties and previous analysis results. In those cases where analysis of a component depends also on factors outside the component, special care must be taken to identify to what extent the reused information is still applicable in the new environment.

Finally, through the integration of nLight (see Chapter 5), PRIDE makes extra-functional property a first class citizen of the development and facilitates their seamless management. Elements of the software architecture can be enriched with a collection of structured attributes such as behaviour and resource models, dependability measures, timing properties. Additionally, users can contribute to the pool of extra-functional properties available in PRIDE in registering new user-defined attribute types.

7.3 Overview of PRIDE

Based around ProCom and the described overall approach, we have developed several tools and tightly integrated them together to create PRIDE. PRIDE is built as an Eclipse RCP application that can be easily extended through the addition of new plugins. As shown in Figure 7.6, the core part of PRIDE currently consists of a component explorer, component editors, an attribute framework (nLight), an analysis framework and a synthesis tool. PRIDE can be extended by adding new extra-functional properties (attribute definitions) together with

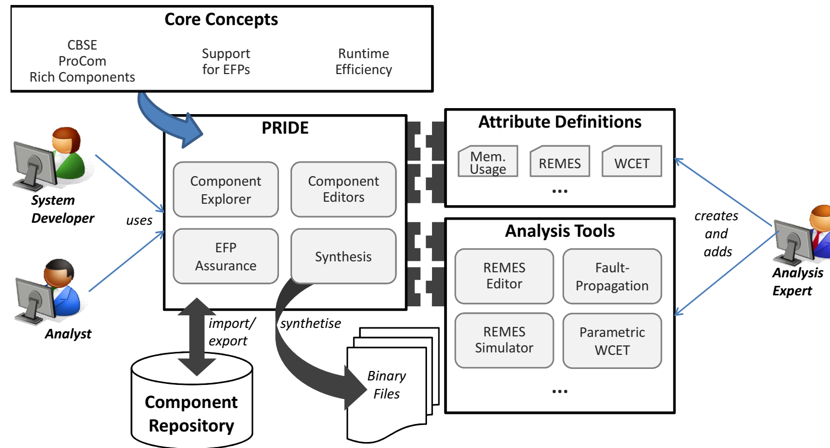


Figure 7.6: Architecture of PRIDE.

their corresponding analysis support when needed. Figure 7.7 shows a screen-shot from PRIDE, with some of these parts highlighted.

Component Explorer The component explorer enables browsing the list of components available in the current development project. In it, a component owns a predefined and extensible information structure that corresponds to the aforementioned rich component concept. The component explorer also supports component versioning, and importing and exporting of components from a project to a component repository, making them available for reuse in other projects.

Component Editors The component editors are used for developing an architectural model of components and a system as a whole. They are built around the ProCom component model and represent one of the central parts of PRIDE. Components from both ProCom layers are treated in a uniform way. The component editor provides two independent views on a component, *external* and *internal view*. The *external view* handles the component specification, including information such as the component name, its interfaces and possibly extra-functional properties. The *internal view* focus on component internal structure implementing its functionality and it depends on the component realization type. For composite components, the internal view corresponds to a

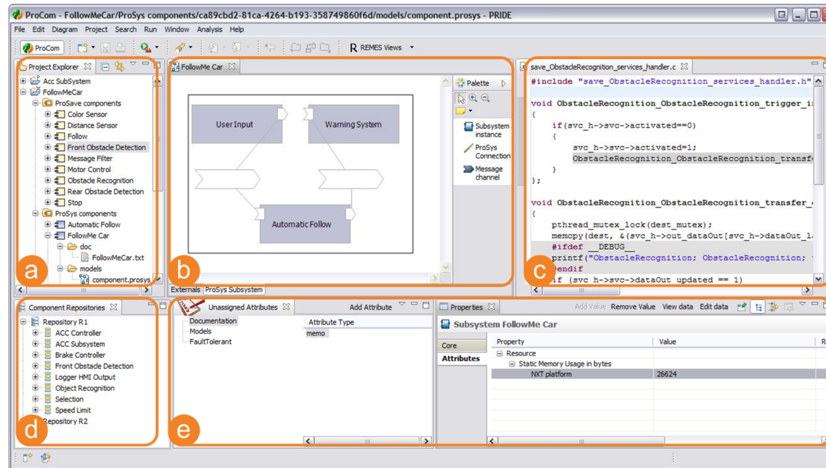


Figure 7.7: A screenshot of PRIDE showing *a)* the component explorer; *b)* a component editor; *c)* a code editor; *d)* the repository browser; and *e)* the attribute framework.

collection of interconnected subcomponent instances, and a graphical editor is available allowing modifications to this inner structure (e.g., addition/deletion of component instances, connectors and connections). For primitive components, the internal view is linked to the component implementation in form of source code. Editing the component code is facilitated by features such as syntax highlighting and auto-completion, provided through the integration of the Eclipse C/C++ Development Tooling (CDT) plugins.

Extra-Functional Property Assurance The extra-functional properties assurance is realised by the integration of two tools: nLight, the attribute framework described in Chapter 5 and the analysis framework. nLight provides a uniform and user-friendly structure to seamlessly define and manage extra-functional properties in a systematic way. Moreover, it also supports the packaging of the different development artefacts in components. It enables attachment of extra-functional properties, as attributes, to selected architectural elements of the component model. Attributes are defined by an attribute type, and include attribute values with metadata and the specification of the conditions under which the attribute value is valid. One key feature is that the attribute

framework allows an attribute to be given additional values during the development without replacing old values. This allows us to define early estimates for extra-functional properties even before actual architectural element is implemented. Such values can be used for analysis in early stages of system development. Later, when the element is more mature, we can add refined values for extra-functional properties allowing us to conduct more accurate analyses.

The *Analysis Framework* provides a common platform for integrating in a consistent way various analysis techniques, ranging from simple constraint checking and attribute derivation (e.g., propagating port type information over connections) to complex external analysis tools. Analysis results can either be presented to the user directly, or stored as component attributes. They are also added to a common analysis result log, allowing the user easy access to earlier analysis results.

Through the use of extension points in the analysis and attribute frameworks, PRIDE provides support to easily integrate new analysis techniques together with their associated extra-functional properties. The analysis techniques already integrated in PRIDE include parametric component-level worst-case execution time analysis [83], model checking of behavioural models [84], and fault-propagation [85].

Synthesis The synthesis part of PRIDE automates the generation of interfaces for primitive components in the lower layer, and generation of code for composite components in both layers. It also produces build configurations (in debug and release mode) for each level of composition.

Based on models of the physical platform and the allocation of components to physical nodes, the synthesis also produces the binary executable files of each node in the system [86]. The synthesised code relies on a middleware that has been ported to different platforms, including POSIX-compliant operating systems, FreeRTOS and JSP.

7.4 Summary

In this chapter, we have described two integrated development environments supporting a component-based development approach for building embedded systems. The first IDE, the Save-IDE, a prototype based on the approach prescribed in SaveCCT, allowed us to get valuable inputs for the development of PRIDE. In particular, many concepts that have been introduced in PRIDE are based on the experiences gained from the development and use of the

Save-IDE. These concepts include for example having components as the main development units, enforcing the separation between component type and instances at design, providing a flexible and multiple steps synthesis and lifting the importance of extra-functional properties during the development.

PRIDE is based on an architecture relying on ProCom components with well-defined semantics that serve as the central development entity, and as means to support and aggregate various analysis and verification techniques throughout the development — from early specification to synthesis and deployment. Through the use of nLight, PRIDE also provides generic support for integrating extra-functional properties into architectural elements and systematically managing them in an uniform way.

In addition to complement PRIDE with new modelling and analysis techniques, and additional extra-functional property specifications, an interesting future work would be to investigate how PRIDE could be extended to supported multiple, possibly distributed, users. In this way, PRIDE would also enable distributed component-based development of embedded systems.

Chapter 8

Extended Examples

Applying research results in practice provides valuable insights on contributions. Among others, it facilitates discovering their advantages and limitations. This is the main purpose of this chapter which investigates through three examples:

1) the integration of analysis techniques based on dedicated models into component models through ProCom and nLight,

2) the inheritance of extra-functional property values between component type and component instances, and

3) how nLight can be used in practice through the development.

Due to limitations of PRIDE and nLight, some of the aspects presented of this chapter have been realised outside these tools such as the synthesis and analyses. Additionally, for clarity purpose, the architectural designs and the excerpts of extra-functional properties are not illustrated through screenshots but have been recreated based on the original artefacts.

8.1 The Turntable

In this section, we evaluate how ProCom can be combined with analysis models through nLight. This evaluation is based on the turntable drilling system by Bos and Kleijn [87] and Bortnik et al. [88], and the use of REMES [74] as a representative analysis model. REMES is a language for high-level formal

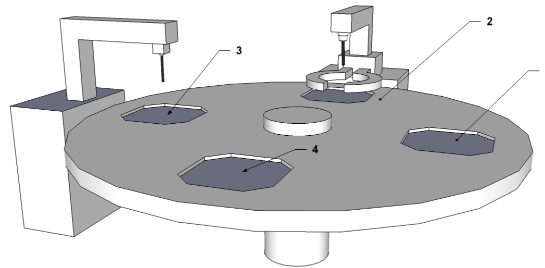


Figure 8.1: The turntable system (load and unload stations are not shown), illustration courtesy of Jan Carlson.

behaviour modelling that allows modelling the behaviour of individual components in terms of functionality, timing and resource usage. In turn, this permits analysing system level properties, while also supporting reuse of behavioural models when components are reused. It also illustrates the use of models as a special type of extra-functional properties.

8.1.1 Overall System Description

The system, depicted in Figure 8.1, consists of a rotating table that moves products between processing stations where they are drilled and tested. Four types of processing stations are involved in the turntable drilling system: a load station, a drilling station, a testing station and an unload station. For clarity purpose, load and unload stations are not shown in Figure 8.1.

The load station places new products on the table (1), after which they are moved to the drill station (2) by rotating the turntable $90°$. Drilling requires that the product is securely held in place by a clamp mechanism. After drilling, the product is moved to the testing station (3) where the depth of the drilled hole is measured. Finally, the unload station (4) removes the product from the table, provided that it passed the test. If not, it remains on the table to be drilled and tested again. The turntable has four slots, each capable of holding one product. Thus, the stations can operate in parallel, so that while the first piece is being tested, a second piece can be drilled, etc.

System Requirements

From the system requirements, we focus on the following:

- *Requirement 1:* The system must be deadlock free.
- *Requirement 2:* A product must be clamped when drilled.
- *Requirement 3:* The table should never turn when one of the stations is operating
- *Requirement 4:* Processing five products should never take more than 25 seconds (assuming at most one failed drilling).

In addition, we want to address the following question:

- "What is the minimum energy consumption for processing five products?"

8.1.2 Architecting the Turntable in ProCom

Since the different stations are relatively independent, we model each station, and the turntable, with a separate component. Accordingly, we define the *Loader*, *Driller*, *Tester*, *Unloader* and the *Turntable* subsystems in ProSys. In order to achieve synchronization between the stations and the table, e.g., guaranteeing that the table turns only when no processing station is operating, as expressed in requirement 3, and that only products which pass the test are unloaded, we define an additional subsystem: the *Controller*.

Next, the interfaces of these identified components need to be specified. Since the component model has been imposed, the available communication mechanisms between components are restricted to asynchronous message passing for the active and independent parts of the system (synchronous control- and data-flows are available only in the lower layer). At this step, it is possible to browse a component repository to find pre-existing components which functionalities and possibly extra-functional properties that match the requirements. In the case of our turntable system, we assume that the *Loader* and *Unloader* components can be reused from a previous project. For the remaining components, i.e. Drilling Station, Tester station, Turntable and Controller, the interfaces remain to be specified. Figure 8.2 illustrates the component interfaces and shows how components are connected together.

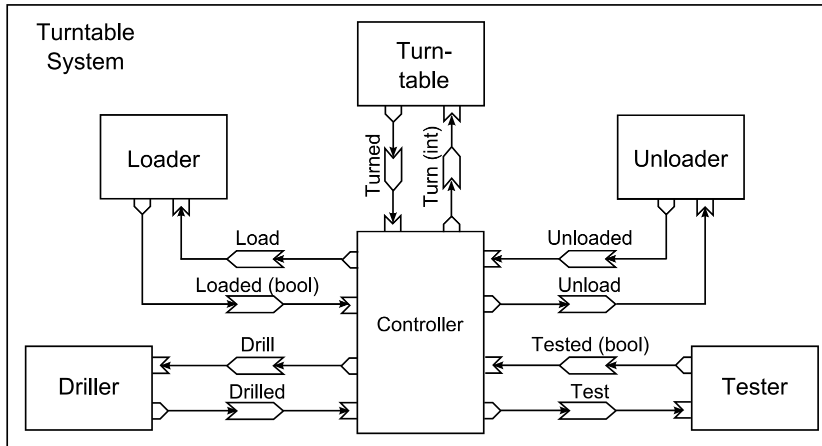


Figure 8.2: ProCom design of the turntable system.

The *Turntable* component receives a message when the table should be rotated. In order to make the component reusable in different systems (e.g., a turntable with more than four stations), the angle of rotation of the table can be specified in the message. When the table has been turned, a message is sent to inform other parts of the system.

The *Tester* and *Driller* have similar interfaces; an incoming message telling the station to start processing, and an outgoing message indicating that it has finished. The output message of *Tester* also contains a boolean value representing if the test succeeded or not.

The *Controller* keeps track of the current status of the four slots, and activates stations accordingly, by sending messages to each stations and receiving messages back once they are done. Consequently, the interfaces of *Controller* must be compatible with the interfaces of the stations (including those of the reused *Loader* and *Unloader*).

It is possible to further decompose each of the ProSys components specified in the architecture design. According to the level of complexity of the functionality and the potential for distribution, each component can be decomposed into smaller ProSys components or alternatively into ProSave components. However before doing that, the developer may want to validate first the feasibility of the design proposed so far. Some properties can be directly analyzed from the ProCom design alone, such as verifying conformance between connected ports and channels. To reason about requirements such as the ones identified

in Section 8.1.1, it is typically necessary to use a dedicated formalism possibly supported by various analysis techniques.

For this extended example, we exemplify through the use of REMES how such formalism can be integrated into ProCom to facilitate analysis of selected extra-functional properties.

8.1.3 Attribute Type Identification and Specification

The first step towards integrating analysis of functional and extra-functional properties into ProCom is to identify the set of required attribute types. It corresponds to the functional and extra-functional properties needed to satisfy the system requirements extracted in Section 8.1.1 plus all the artefacts required or produced during the development process, including the analysis models and the analysis results. In case no suitable specification is available in the attribute registry, necessary attribute type specifications must be created and registered. Table 8.1 presents a non-exhaustive list of attribute types from nLight that can be used in the context of the turntable drilling system.

From a reuse point of view, it is convenient to have a dedicated attribute type for REMES. Through the use of the REMES attribute, it is possible to package a REMES model with the component type which it formally specifies the behaviour of. This facilitates reusing the component and its corresponding analysis model. According to the REMES attribute type specification, REMES attribute instances can be attached to both ProSys subsystems and ProSave components, and have complex attribute values consisting of *(i)* a reference to the relative position of the REMES model in the component structure and *(ii)* a reference to the relative position of a file containing the mapping between the component's ports and the variables used in the REMES model.

Furthermore, in order to provide analysis results, REMES models are transformed into Priced Timed Automata (PTA) models that are analysed with the UPPAAL model-checker [78]. If PTA models can be easily and efficiently obtained from REMES models then there is no need to package them in the component. Consequently, the corresponding *PTA* attribute type is not necessary. On the other hand, if this translation takes time or the models need to be regenerated often, it can be useful to store them instead and make use of the *PTA* attribute type. For the purpose of this example, we assume the latter option.

The *deadlock free*, *minimum energy consumption* and *maximum energy consumption* attribute types are example of "simple" extra-functional properties that can be extracted from analysing the models.

Table 8.1: Attribute type specification for the Turntable System

TypeID	Attributables	Data Format	Support Mechanisms		Short Documentation
			Viewer	Editor	
REMES	Component	<Path; Path>	REMES Editor	REMES Editor	Reference to a REMES model, with: Path1 corresponding to the relative path to the REMES model, and Path2 to the relative path to the mapping file
Priced Timed-Automata	Component	Path	PTA Editor	PTA Editor	Reference to a priced timed automata model
Deadlock Free	Component, Instance, Service	"Yes" "No" "Unchecked"	Inline	Inline	Whether for all reachable states, there exists some path to a quiescent state.
Minimum Energy Consumption	Component	Float	Inline	Inline	the Minimum amount of energy drawn from the supply during a single clock period
Maximum Energy Consumption	Component	Float	Inline	Inline	the Maximum amount of energy drawn from the supply during a single clock period

8.1.4 Early Formal Analysis

In this section, we give a brief overview on the formal analysis performed for the Turntable system based on the use of REMES behaviour models.

Behaviour Modelling in REMES

We model the functional, timing and resource usage behaviour of the turntable components as REMES models. Since the *Loader* and the *Unloader* components are supposed to be reused, we assume that they already have their own behavioural model. For each of the remaining components, a REMES models is created. Given that this case study focuses on integration of analysis techniques in component models and not on the analysis part itself, we only present here the REMES models of *Driller* and *Controller*, depicted respectively in Figure 8.3 and 8.4.

The *Driller* component is responsible for moving the drill up and down and for locking and unlocking the clamp. In order to do this, it reads values from the drill and clamp sensors, modeled by boolean variables *sdu* (drill in upmost position), *sdd* (drill in downmost position), *scl* (clamp fully locked) and *scu*

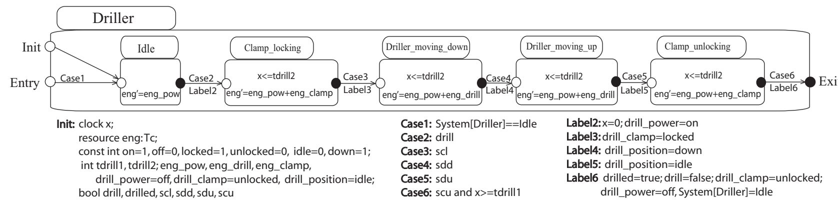


Figure 8.3: The Driller modeled in REMES.

(clamp fully unlocked). Neither of the two message ports of *Driller* carries values, and thus they are mapped to two boolean variables *drill* and *drilled*.

The *Driller* remains in the *Idle* mode until receiving a *drill* message. When this happens, the component goes through a sequence of submodes: *Clamp_locking*, *Driller_moving_down*, *Driller_moving_up* and *Clamp_unlocking*. Each of these submodes is exited as the result of a sensor value turning *true*. When exiting the last submode, a *drilled* message is sent, indicating that the operation is finished.

REMES model also enables modelling the consumption of energy of the *Driller* subsystem. We assume the following: powering the *Driller* consumes *eng_pow* units of energy per time unit, locking or unlocking the clamp consumes *eng_clamp* units of energy per time unit and drilling consumes *eng_drill* units of energy per time unit. Moreover, we assume that the time of each *Driller* operation cycle is bounded to the interval *[tdrill1, tdrill2]*.

The *Controller* component, depicted in Figure 8.4, keeps track of the states of the four slots and operates the stations and the turntable accordingly by exchanging messages with all of them. The behaviour defined by the REMES mode consists of two main submodes, one in which the controller waits for messages from the stations, and one waiting for the turntable to finish turning.

The submode *Wait_for_turning* is exited when the *turned* message arrives. Depending on the current state of the four slots, messages are sent out to the respective station. This is managed by the four conditional connectors and the guards (*Case9*, ..., *Case16*). For example, the *load* message is only sent if the first slot is empty, and the *drill* message is only sent if the second slot is occupied. The local variables *signal_loader* etc. are used to keep track of what messages were sent. When all messages are sent, the history variable *System[Controller]* is assigned the value *Wait_for_stations*. Thus, the *Controller* will continue executing in that submode when reentered.

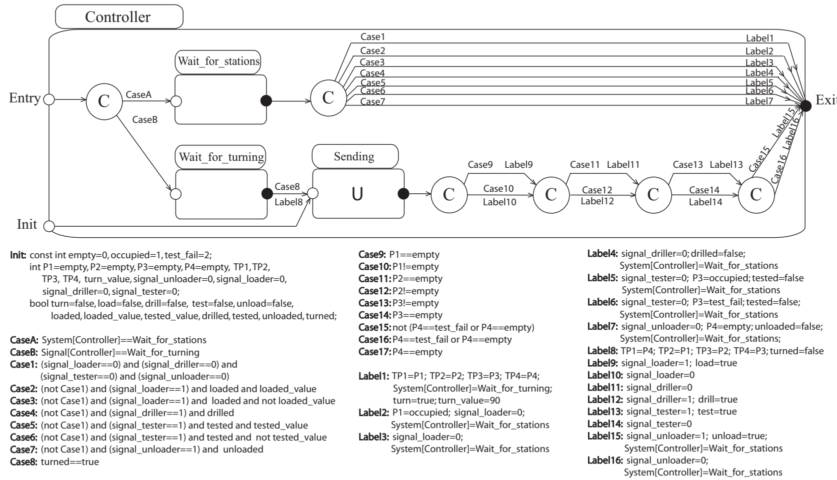


Figure 8.4: The Controller modeled in REMES.

In submode *Wait_for_stations*, the *Controller* waits until it receives a reply to one of the sent messages. Since this is a non-lazy mode, it must be exited as soon as the guard on one of the outgoing discrete actions (*Case1*, ..., *Case7*) is satisfied. If the message carry a value (which is the case for *loaded* and *tested*), it is used to update the state of the corresponding slot. When all messages have been received, the message *turn* is sent to the *Turnable*, and the history variable is set to *Wait_for_turning* before exiting, meaning that the execution will be resumed in that submode.

Results From Applying REMES to ProCom

Once a REMES model has been created for each ProCom component, the turntable drilling system can be formally analysed. To do so, the system is first transformed into a network of priced timed automata (PTA) models. In the example, the semantic translation from REMES to PTA is done manually, as described in [74], although ideally this should be automated.

Next, the design of the system is verified against the identified system requirement that are expressed as temporal logic formulas. Table 8.2 maps the system requirements from Section 8.1.1 together with their corresponding temporal logic formulas and verification results.

Table 8.2: System properties and verification results.

Req.	System property	Temporal logic formula	Verification result
1	The system should be free from deadlocks.	*A[] not deadlock*	Satisfied
2	A product must be clamped when drilled.	*A[]* Driller.Driller_moving_down *imply* Driller.drill_ clamp==locked	Satisfied
3	The table should never turn when one of the stations is operating.	*A[]* (Turntable.Turn1 *or* Turntable.Turn2) *imply* (Loader.Idle *and* Unloader.Idle *and* Tester.Idle *and* Driller.Idle)	Satisfied
4	Processing five products should never take more than 25 seconds (assuming at most one failed drilling).	*A[]* (*not* loaded_failed *and* time>25 *and* failed_products≤1) *imply* processed_products≥5	Satisfied
5	What is the minimum energy consumption for processing five products?	*E*⟨⟩ (processed_products==5)	14 300 units[1]

Property 1 from Table 8.2 is a generic safety property, specifying the absence of a system deadlock, i.e., the system cannot come to a state from which it cannot continue operating. The turntable system is verified to be deadlock free. The next step is to verify that it satisfies the functional system requirements, here represented by properties 2 and 3. Properties 4 and 5 are examples of extra-functional properties addressing time and resource usage, respectively.

8.1.5 Attribute Instance Creation

Relying on the previously described activities, i.e. the creation of the ProCom components, their behavioural modelling, the analysis and the availability of suitable attribute types, it is now possible to associate information about functional and extra-functional properties with the newly defined components. Accordingly, several attribute instances are added to each ProCom components.

First, an instance of the REMES attribute type is added to each ProCom component as illustrated in Figure 8.5 for the Controller component. This instance associates with the component, its REMES behaviour model and the corresponding mapping file which specifies the correspondence between the

[1]This value is extracted from UPPAAL's execution traces.

variables used in REMES model and the component's ports. Both files are physically located in the file structure of the components as visible in the description of the attribute instance value.

Additionally, for each priced timed automata model derived from a REMES model and results of the performed analysis, a PTA, deadlock free, minimum energy consumption and maximum energy consumption attribute instances, are inserted into the corresponding component specification through nLight. This figure also shows the use of nLight for the packaging the various development artefacts within a component.

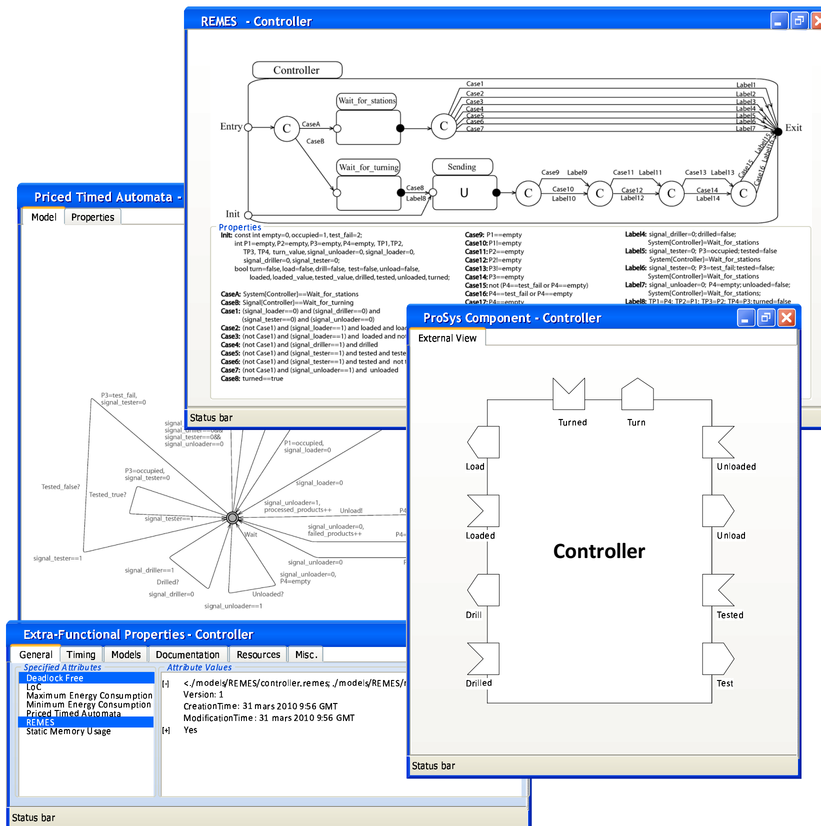


Figure 8.5: The ProSys Controller Component packaged with its behavioural models (REMES and PTA) through nLight

8.2 The Personal Navigation Assistant System

In this section, we explain the mechanisms of extra-functional property refinement proposed in Chapter 4 on a personal navigation assistant system that will be designed as a ProCom system and enriched with extra-functional properties.

8.2.1 Overall System Description

A Personal Navigation Assistant (PNA) relies on the Global Position System (GPS) to provide aid to navigation functionalities such as computing the best routes between two cities, distance and time to arrival, current speed, direction, etc. The GPS system is composed of 24 Earth-orbiting satellites periodically sending information to GPS receivers that calculate their Earth-based geolocation. In common language, GPS refers to the GPS receiver devices only. Likewise in this section, we focus on the GPS receiver part of the PNA.

A GPS receiver is a device able to determine its location on Earth through a trilateration calculation method that requires the exact position of at least three satellites. With three satellites, a GPS is able to estimate its 2D-position (longitude and latitude) whereas with four satellites, it can also compute its altitude. The more satellite positions the receiver get, the more accurate is the position calculation. For example, most of today receivers, such as the Garmin G18 [89], tracks simultaneously up to twelve satellites for better results. Other type of receivers includes multiplexing channel receiver that can only follow one satellite at a time, thus forcing them to switch rapidly between the satellites being tracked at the cost of time and precision.

In order to know the satellite's position precisely, the GPS receiver must be fully aligned with the signal of the satellite being tracked. To enable satellite's position discovery, the GPS receiver uses a clock to have the current time and an almanac containing the supposed positions of a satellite at a given time.

To create the PNA, the GPS receiver is associated to a navigation processor (Navigation System) that computes the navigation data (current position, direction, current speed, etc.) and a graphical user interface that enables displaying the device's geolocation data on maps, together with the navigation data and other information such as distance and time to destination, point of interests, etc.

8.2.2 Architecting the PNA in ProCom

To comply with the previous description, a PNA system (illustrated in Figure 8.6) is developed out of four ProSys components (the *GPS Reveiver*, *Power Management*, *Navigation System* and *UI*) since a PNA installed in a car could be distributed, i.e. having its central computation unit in one part of the vehicle while the signal receiver units would be located closer to the roof for better reception, also with respect to the parallel activity.

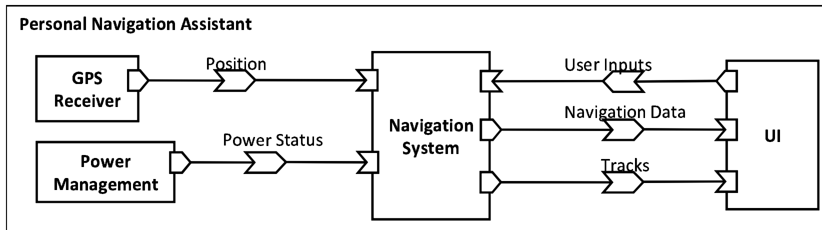


Figure 8.6: a PNA system modelled in ProCom

Looking closer at the *GPS Receiver* component shown in Figure 8.7, it is a composite ProSys component that consists of the *Clock* and *Almanac Store* ProSys components to help the GPS receiver to faster locate the satellites on start-up and a *Parallel Receiver* component that simultaneously tracks up to twelve satellites to compute the geolocation of the device.

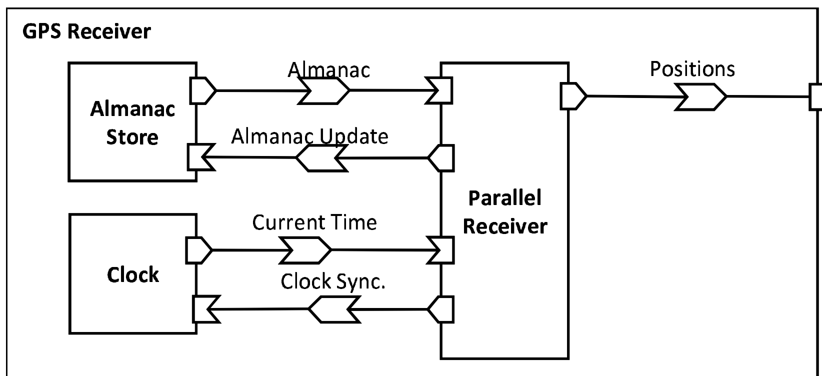


Figure 8.7: Model of the GPS receiver as composite ProSys components

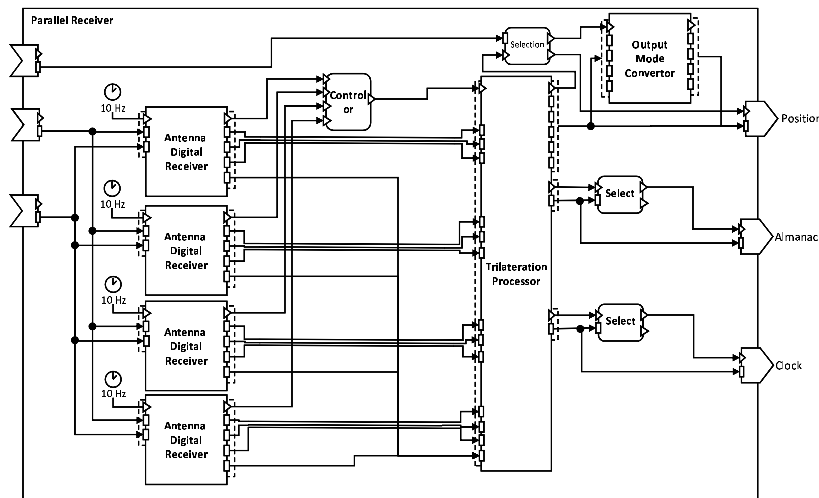


Figure 8.8: A simplified version of a ProSys primitive receiver

The *Parallel Receiver* is a primitive ProSys component built out of ProSave components as shown on Figure 8.8. It consists of twelve instances of an *Antenna Digital Receiver* ProSave component, a *Trilateration Processor* and an *Output Mode Converter*. For readability purpose, only four instances of the Antenna Digital Receiver are depicted on Figure 8.8.

The Antenna Digital Receiver is in charge of the synchronization with the satellite's signal and get the satellite location. The Trilateration Processor computes the actual position of the devices and if activated, the Output Mode Converter converts the position into a different format. The communication between these components follows a pipe-and-filter architectural style separating data flows from control flows. Data input and output ports are denoted by small rectangles whereas trigger ports are triangles. Moreover, the antenna digital receivers are periodically activated every ten seconds. Once one of the Antenna Digital Receiver has terminating its computation, it activates the trilateration processor.

8.2.3 Attribute Type Specification

From the above description and the specification of the Garmin G18 [89], several extra-functional properties emerge as important to consider when developing a global positioning system.

One of them is the response time, which can be defined as the time to react to a given input. In a GPS, this corresponds, for example, to the time needed to inform the user of his/her physical location. Accordingly, an attribute instance corresponding to a response time extra-functional property is meaningful between an input port and a set of output ports or a service. Yet, given that the main purpose of this section is to evaluate the mechanisms of inheritance of extra-functional properties between component type and component instances, we, instead, associate response time with the component type and instances.

We use the same reasoning for the specification of related extra-functional properties: the acquisition time that specifies the amount of time that it is required to correctly receive the satelitte's position signal, once the position of the GPS satellite has been found; the searching time that, on the other hand, corresponds to the time required to search and acquired the GPS satellite signals; and the processing time, which is defined as the time required for the receiver to compute the position.

Other properties that can be useful during the development of a GPS include the vendor name, worst-case execution time and static memory usage. Table 8.3 summarizes the specification of the attribute types corresponding to these properties.

Inheritance Policies

In addition, the usage of each attribute type can be constrained by the definition of inheritance policies. These inheritance policies are used to specify whether an attribute value specified on a component type is available to its component instances, and, if the attribute value is available, the rules which govern its possible refinement.

Table 8.4 lists the inheritance policies used in the PNA example. For instance, for an attribute type such as "Vendor Name", it is sufficient to have a value for the component type. Having this information on the component instance would simply be redundant in that particular case. As a consequence, the inheritance policy for the attribute type "Vendor Name" is defined as *notInherited*. On the other hand, it is beneficial to be able to inherit attribute values defined on component types for attribute types such as acquisition time, response time and WCET. Using information from the design of the composite

Table 8.3: Attribute type specification for the PNA System.

TypeID	Attributables	Data Format	Short Documentation
Acquisition Time	Component, Instance	Int	The amount of time (in ms) to receive the first information from a satellite
Response Time	Component, Instance	Int	The amount of time (in ms) to react to a stimuli
Vendor Name	Component, Instance	String	Name of the component's producer
Static Memory Usage	Component, Instance	Int	The amount of memory (in kb) statically allocated
WCET	Service	Int	The maximum number of clock cycles a service uses before terminating

Table 8.4: Inheritance policies used in the PNA system.

Identifier	Inheritance Policy	Constraint
Vendor Name	notInherited	none.
Acquisition Time	override	originalValue $\geq$ refinedValue.
Response Time	override	originalValue $\geq$ refinedValue.
WCET	inherited	originalValue $\geq$ refinedValue.
Static memory	final	none.

component in which the component instances are used, analyses can provide more accurate values. Accordingly, we set the inheritance policy for these attribute types to *override*, which means the value is inherited from the component type to the component instances and that this value can be refined. In addition, we enforce that if the value is refined in the component instances, this value should be smaller than the original value. This is done through the constraint: "$originalValue \geq refinedValue$". The attribute type static memory is set as *final*. This implies that attribute instances will inherited any static memory attribute instance defined on a component type but this value cannot be modified.

8.2.4 Application on the GPS receiver

Table 8.5 lists the attribute values defined on the GPS receiver component type: one value for vendor name, three values for acquisition time (one requirement and one measurement for a warm start, and one estimation for a cold start), and one static memory value.

Table 8.5: Attribute instances specified for the GPS Receiver component type.

TypeID	Attribute Values
Vendor Name	[-] MDH
Acquisition Time	[-] 50 – Source: Requirement – Comment: Warm Start [-] 40 – Source: Measurement – Comment: Warm Start [-] 450 – Source: Estimation – Comment: Cold Start
Static Memory	[-] 305 – Source: Measurement

Table 8.6 shows how these values have been inherited and refined on the GPS component instance used in the PNA component. Due to the fact that the attribute type Vendor Name is specified as notInherited, the attribute instance defined on the GPS receiver component type is not available for its component instance.

On the other hand, acquisition time and static memory instances are all inherited on the GPS receiver component instance. However, whereas the static memory instance cannot be refined since its inheritance policy is set as final, the acquisition time instances values can be, their inheritance policy being defined as override. The first value for the acquisition time, i.e. the requirement for a warm start, is then constrained from 50s to 43s. After measurement, the second value should be refined to 60s. However, the constraint specified for the inheritance policy dictates that a refined value can be smaller or equal to the one defined on the type only. This raises an alert. Either, the measurement performed on the component instance is incorrect and in this case, 40s should be the value to use, or the value set on the component type was too small and must be increased to at least 60s.

Table 8.6: Attribute instances specified for the GPS Receiver component type.

TypeID	Attribute Values
~~Vendor Name~~	[-] ~~MDH~~ Vendor Name is notInherited
Acquisition Time	[-] ~~50~~ 43 Acquisition Time is override – Source: Requirement – Comment: Warm Start [-] ~~40~~ 60 Constraint not respected – Source: Measurement – Comment: Warm Start [-] 450 – Source: Estimation – Comment: Cold Start
Static Memory	[-] ~~305~~ 305 Static Memory is Final. Cannot be refined – Source: Measurement – Platform: Linux

8.3 The Automatic Driving System

In this section, we demonstrate the conjoint use of ProCom and nLight for the development of a representative example of embedded real-time systems. The choice of the example has been guided by the following rationales:

- *Rationale 1:* The subject of the study must be small yet representative of an embedded real-time systems (resource constrained ECU with at least one sensor and one actuator).
- *Rationale 2:* The subject should enable reasoning about typical extra-functional properties of embedded systems such as execution time, and memory usage.
- *Rationale 3:* The subject of study must enable reusing components in different applications.
- *Rationale 4:* The subject of study must enable porting the application between different hardware.
- *Rationale 5:* A real-time operating systems should be used.
- *Rationale 6:* A predictable programming language should be used, C (or C++) preferably.

Based on these rationales, we have accordingly defined the "automatic driving system" example in section 8.3.1 and, in the subsequent sections, we describe a part of its development focusing on the modelling and analysis stages. An important part of the example is also dedicated to explain the use of multi-valued context-aware extra-functional properties during the development of the system.

8.3.1 Overall System Description

The automatic driving system is a "drive-by-wire"-like solution inspired by the "Distance Control Assist" system proposed by Nissan [90], which main purpose is to enable to electronically assist the driver to maintain a safe distance to a preceding vehicle.

In the hardware specification of the Nissan's "Distance Control Assist" system, illustrated in Figure 8.9, the system is composed of a main controller in charge of deciding the behaviour of the vehicle according to the current situation. In order to do this, the controller is connected to a radar sensor to estimate

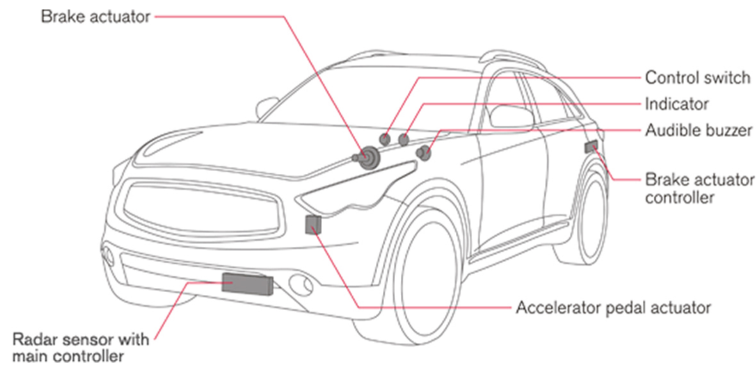


Figure 8.9: Hardware for the Distance Control Assist system from [90]

the distance of a vehicle in front, a pedal actuator to control the acceleration, a brake actuator to control the braking. The driver can activate the distance control assist system through a control switch button, and once activated, the driver will be informed of the necessity of braking by an indicator and a buzzer signal. If the driver interferes (for example in braking or accelerating manually), the distance control assist system is deactivated. In that case, the car is driven manually again.

The hardware and overall behaviour of the automatic driving system is similar to the distance control assist system. However, instead using a real vehicle as target platform we use the Robotic Command eXplorer (RCX brick as short). Both platforms provide the same facility to build embedded systems out of a controller that interacts with the physical world through sensors and motors. The hardware characteristics of these different parts are listed in Table 8.7.

Furthermore, the development of the system is broken down into the two iterations: *Iteration 1* in which a software system is developed to electronically drive a vehicle based on user's inputs and *Iteration 2* in which the software developed in iteration 1 is enhanced with additional functionalities and hardware to enables the vehicle to be driven in an autonomous way in following a preceding vehicle at a safe distance.

Table 8.7: Specification for the RCX Platform

CPU	Hitachi Renesas H8 series (H8/300) H8/3292 16Mhz
ROM Internal RAM External RAM Additional Storage	Total: 16kb Available: 10kb (6kb used by the OS) 51kb 32kb None
Sensor ports Available Sensors	3 Light sensor Touch sensor
Buttons Button Type	4 on/off, program, view, run
Actuator ports Motors	3 2 at 360 RPM
Display	5-segment LCD
Speaker	1
Communication	bidirectional IR
Timers	built-in 10Mhz
RTOS	BrickOS

Iteration 1: Manual driving

The purpose of this iteration is to develop the basic part of the system for which extra-functional properties can be assessed during the development process, hence demonstrating how multi-valued context-aware extra-functional properties can be specified and how their values are used and refined during the development process.

In iteration 1, the system simply enables to manually drive the vehicle forward, backward, left, right, to accelerate, decelerate and also honk. Since there is no physical support to manually drive the car, the car is instead driven remotely through wireless communication. The vehicle is equipped with a back parking sensor which signals whether the car touches an obstacle while backing. In that case, a sound-alarm is raised and the car stops. A display informs the driver about the current direction (back, forward, paused) together with the speed of the vehicle. In short, in iteration 1, the system behaves as summarized in scenario 0.

- *Scenario 0: Manual driving*: The driver is fully in charge of the vehicle (except when backing in presence of an obstacle).

The hardware for the platform of the vehicle is built out of a Lego RCX brick that uses brickOS as real-time operating system, two motors to control the wheels, a speaker, a display, four buttons, a bidirectional IR communication device, a touch sensor positioned in the back of the platform. Additionally, a remote control is used to manually drive the car.

Iteration 2: Autonomous driving

In this iteration, the system is enhanced with a “Autonomous Driving System” (ADS) feature, which enables the car to automatically follow a target at a specified distance such as twice the braking distance by default. If the driver interferes with the driving in accelerating or braking for example, the system goes back in manual driving. The ADS is available in two versions: a low-end and high-end versions. In the low-end version, it is assumed that the target vehicle can move back and forth only and cannot make turns. As in previous iteration, when in presence of an obstacle the vehicle equipped with the ADS system must signal when it cannot back further and stops. The distance is calculated through a distance sensor. In the high-end version, the target vehicle can also turn. In that case, the vehicle equipped with the high-end version of ADS behaves as the low-end version with the difference that it can lose the target vehicle. As a result, the vehicle equipped with the ADS needs to be able to relocate the target before pursuing its tracking.

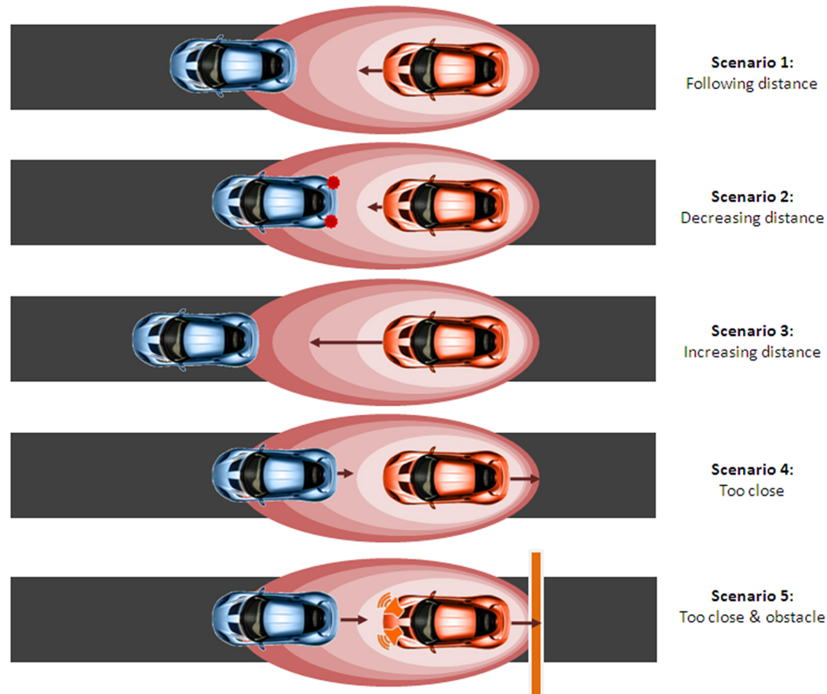


Figure 8.10: The different driving scenarios

As illustrated in Figure 8.10, five basic driving scenarios are envisaged for the ADS system in addition to scenario 0 from iteration 1 which is the default scenario.

- *Scenario 1: Following.* This is the default scenario for the ADS. Once the automatic mode has been selected, the vehicle equipped with the ADS system first wait for a target to follow. Once a target is at detection range, the vehicle starts following the target.

- *Scenario 2: Decreasing/increasing speed of the target.* If the distance to the target decreases or increases, the vehicle adapts its speed to the target

- *Scenario 3: The target is backing.* If the target vehicle is backing (i.e. the distance to the target continue to decrease although the vehicle has reduced the speed or stopped), the vehicle equipped with the ADS system backs too.

- *Scenario 4: Obstacle when the target is backing.* While in Scenario 3, the vehicle equipped with the ADS system detects an obstacle when backing, the vehicle then stops and an alarm signal will sound.

- *Scenario 5: Driver's intervention.* If the driver interferes in the automatic driving with braking or accelerating for example, then the system changes to manually driving (see scenario 0). Note that the interval between the user action and its effect should be of 215ms at most (this corresponds to the average reaction time for a human).

The hardware used in this iteration is the same hardware as in iteration 1, with the difference that a light sensor is added in the front of the vehicle to estimate the distance to the target platform.

8.3.2 Attribute and Metadata Type Specification

Similarly to what has been done in the Turntable and GPS examples, it is necessary to identify the set of attribute types that should be considered during the development of the system. In case no suitable specification is available in the attribute registry, additional attribute type specifications must be created and registered.

In order to better understand the behaviour of the system, models can be used such as a UML statechart diagram. Similarly to what has been done with REMES model in the Turntable example, a dedicated attribute type can be created. From the availability of this attribute types, the corresponding model can be packaged together with the component it depicts.

For the extra-functional properties aspects, when considering the hardware specification and overall behaviour of the system described in the previous section, timing and memory usage properties emerge as important requirements. In particular, the response time of the drive-by-wire vehicle should be at least equivalent to the average response time of a human driving the car, that is, 215 ms. One of the factors influencing the response time is the execution time. As a consequence, for the purpose of this example, we will consider the *worst-case execution time*. As listed in Table 8.8, the worst-case execution time is the longest execution time that could be observed when the service is executed on its target platform. It is worth noting that the SI base unit should be used as the reference unit. However, in the context of this example, the timers are cadenced at 10MHz, which implies the timing values are in the order of magnitude of the nanosecond. For the sake of clarity, we then express the worst-case execution time in nanoseconds.

Additionally the amount of memory available, both ROM and RAM, is limited as shown in the RCX specification table 8.7. Thus, it is important to evaluate, through the development process, extra-functional properties such as *static memory usage* and *physical size*. The static memory usage corresponds to the maximum amount of stack space that is used by a component to store the temporary data that are necessary for its execution. The physical size corresponds to the size occupied by a component once compiled.

Table 8.8 presents the list of attribute types registered in nLight that are used during the development of the automatic driving system example.

Table 8.8: Attribute type specification without the support mechanisms

TypeID	Attributables	Data Format	Short Documentation
WCET	Service	Int	the longest execution time in nanoseconds that could be ever be observed when the service is executed on its target hardware.
Static Memory Usage (stack space)	Component	Int	The amount of memory in kb used to store the temporary data used during the component execution
Physical Size	Component	Int	The physical size in kb occupied by a component once compiled.
UML Statechart	Component	Path	UML Statechart diagram specifying the dynamic behaviour of a component.
UML Use case	Component	Path	UML Statechart diagram specifying the set of actions available to the system's users.

Further, the information that will be provided by the attribute instances need to be complemented by suitable information to capture the context in which the corresponding attribute value has been obtained: for example, on which platform, by which method, etc. Table 8.9 presents a non-exhaustive list of metadata that are suitable to use. The *Platform* metadata type is used to specify on which target platform the extra-functional property value has been set. The *Source* metadata type describes the method used to assess the value. The *Analysis Type*, only available for the WCET attribute type, allows refining the type of analysis that has been performed: whereas *guarantee* implies that the value has been assessed with safe margin estimations meaning that

the actual WCET will always been inferior to this value, *estimation* does not provide such guarantee. Often, analysis generates outputs that describe how a particular value has been evaluated. Accordingly, it is important to keep these outputs to corroborate the value. This is the role of the *Analysis Output* metadata type, that allows packaging the analysis results with the component for which the attribute value has been assessed. The metadata type *Comment* enables developers to express comments on the value. Other metadata type not explicitly shown in the table include information related to the creation time, the accuracy of the values or its version, etc.

Table 8.9: Metadata type specification

MetadataID	Desc. Of	Value Format	Cardinality
Platform	*	{ "RCX i1", "RCX i2", "NXT", … }	*
Source	*	{ "*Estimation*", "*Measurement*", "*Simulation*", "*Inherited*", "*Analysis with Bound-T*", "*Early Analysis*, … }	*
Analysis Type	WCET	{ "*Estimation*", "*Guarantee*" }	1
Analysis Output	*	Path	*
Comment	*	Text	*
Author	UML Statechart	String	+

8.3.3 Developing the Drive-by-Wire System (Iteration 1)

System Requirement

Developing a system necessitates first to have a clear understanding of the system boundaries and requirements. Once captured, they can then be thoroughly studied. This process generally leads to the creation of different artefacts such as UML diagrams, algorithms, informal documents, etc.

For instance in iteration 1, we use UML use-case diagram to clarify the interactions of the systems (see Figure 8.11). Further, in order to understand the behaviour of the system during its execution, we model it as the UML statechart diagram provided in Figure 8.12. In it, the system behaviour has two main states: either the system is in "*idle*" state waiting for the driver to start the system, or the vehicle is being driven, i.e. the system is "moving". Taking into consideration the characteristics of a vehicle, the moving state can be decomposed as a concurrent hierarchical state: one for the direction (forward, backward or free wheel), one for the speed (constant speed, accelerate, decelerate), one for rotation (left, right, or straight) and one for the honk. Additionally, when an obstacle is met while backing (moving state in the reverse sub-mode for the direction), the systems passes into the "standby and warn" state until either the direction changes to forward or the obstacle is removed.

Architecting the System in ProCom

To realize the "drive-by-wire" solution, the system is decomposed into five independent and active building blocks designed as ProSys components: the *Communication System*, the *HMI System*, the *Engine System*, the *Alarm System* and the *Controller System*. Figure 8.13 shows these components and how they communicate with each others.

The *Communication system* is in charge of simulating the presence of a driver for the vehicle. The communication system receives instructions from the users through the infrared receiver of the RCX brick, i.e. the communication device. When an instruction arrives, it is analysed and encoded as a message compatible with the command message channel before being sent. Such command includes driving direction (forward, backward, left and right), driving speed (faster, slower), stop and warn. Accordingly, the basic functionality of the communication system component is to wait for driving instructions.

The *HMI System* is the interface between the vehicle and the driver. Information messages from the system are displayed to the user on a dedicated display and the user can also directly interact with the systems through the

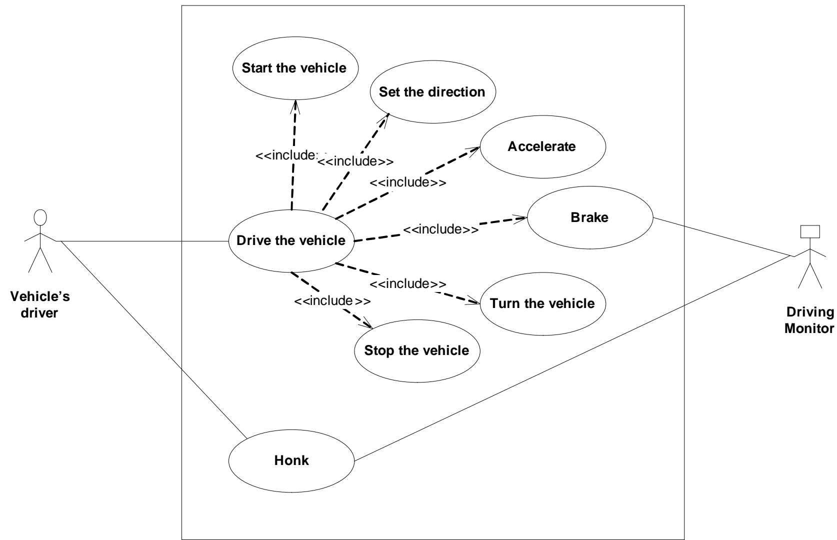


Figure 8.11: Statechart model of the system for iteration 1

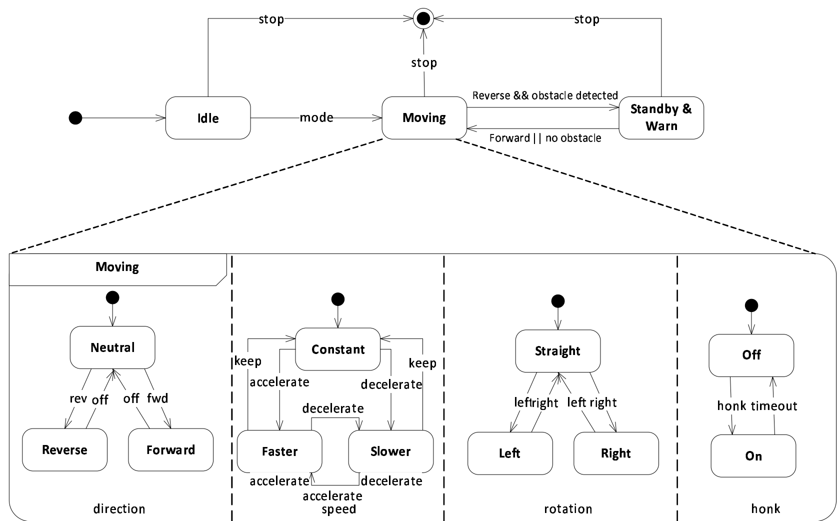


Figure 8.12: Statechart model of the system for iteration 1

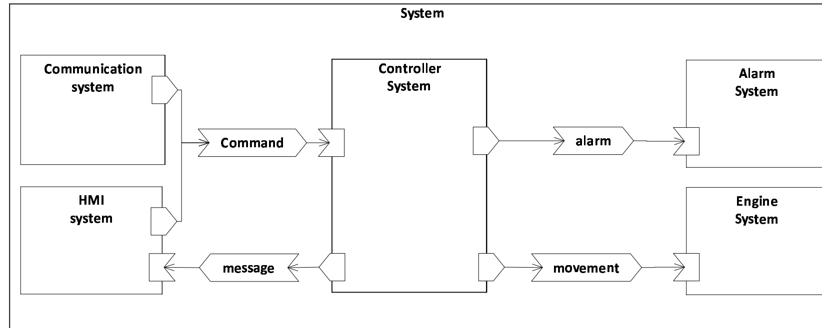


Figure 8.13: High-level description of the system with ProSys components

available buttons. Internally, the HMI system consists of two sub-components: one dedicated to handle the user inputs from the buttons and the other one for displaying information to the user.

The *Engine System* is in charge of controlling the movement of the vehicle. It sets the speed and direction of the vehicle based on desired movement related instructions. The *Alarm System* manages the sound and warning systems to inform when an obstacle is met while backing: when an alarm signal message is received, the alarm system activates the honk for a certain amount of time.

The *Controller System* is the component in charge of taking decision for the whole system. According to the users instructions, it calculates the necessary operational changes that the vehicle must responds to.

Given that, the main focus of the example does not target the distribution and concurrent execution of subsystems aspects proposed by ProCom, we simplify the proposed design in implementing the Engine System and Alarm System as ProSave components in the controller system. This allows on one hand for simplified analyses and assessments of timing and memory properties for the System component, in the sense that the concurrent execution and distribution is not concerned, but on the other hand, this makes the analyses and assessment of these properties more complex for the controller.

Accordingly, the controller system is a ProSys primitive component built out of ProSave Component. Its inner structure is shown in Figure 8.14 and described further in the section "architecting the Controller system with ProSave components".

Setting the Requirements

Now, that a preliminary architecture is available, attribute instances can be used to annotate the ProCom architectural model with information describing the requirements and to package the requirement artefacts produced in the previous requirement analysis phase with their corresponding components.

From the description of the system and the hardware specification, the following requirements can, for example, be extracted:

- **Memory Consumption**

  M1 The software system should fit in 10kb of ROM

  M2 The software system can use at most 32kb of RAM

- **Safety and Timing**

  ST1 While moving, the vehicle must respond, at most, in 215ms.

Table 8.10 shows an excerpt of the attribute values attached to the ProSys components of the architectural model from Figure 8.13. The aforementioned requirements M1 and M2 are set on the System component through the creation of a "physical size" and "static memory usage" attribute instance respectively. ST1 could be captured in a similar way through the creation of an additional attribute type that correspond to a response time extra-functional property.

The requirements set of the System component can be broken down to its individual subcomponents. As observable in Table 8.10, this allows for simple verification of the requirements between the values attached to the sub-components. For example, the Controller has been assigned 6kb of physical memory space. Yet, in deriving a physical size attribute instance for the System component from the values of its sub-components, the total amount of ROM that should be available has become 12kb which is superior to the initial requirements. Accordingly, the attribute instance for the physical size of the controller is refined to pass from 6kb to 4kb.

Architecting the Controller System with ProSave Components

As illustrated in Figure 8.14, the Controller System internally consists of a rear Parking Sensor component, a Decision Center component, an Alarm System component and a Motor Unit component.

Table 8.10: Excerpt of attribute instances concerned with requirements (Platform and Validity Conditions are not visible)

Architectural Element	TypeID	Attribute Values
System	UML Usecase	[-] ./models/UML/uc.uml
	Static Memory Usage	[-] 32 – Source: Requirement – Comment: M2
	Physical Size	[-] 10 – Source: Requirement – Comment: M1 [-] **12** – Source: Derived – Comment: M1 – Comment: Derived from sub-component's requirement values
Controller	UML Statechart	[-] ./models/UML/statechart.uml – Author: Séve – Comment: Approved by Jan
	WCET	[-] 150 000 – Source: Requirement – Type: Estimation
	Static Memory Usage	[-] 10 – Source: Requirement
	Physical Size	[-] ~~6~~ **4** – Source: Requirement – Version: ~~1~~ **2**
HMI	WCET	[-] 30 000 – Source: Requirement – Type: Estimation
...	...	...

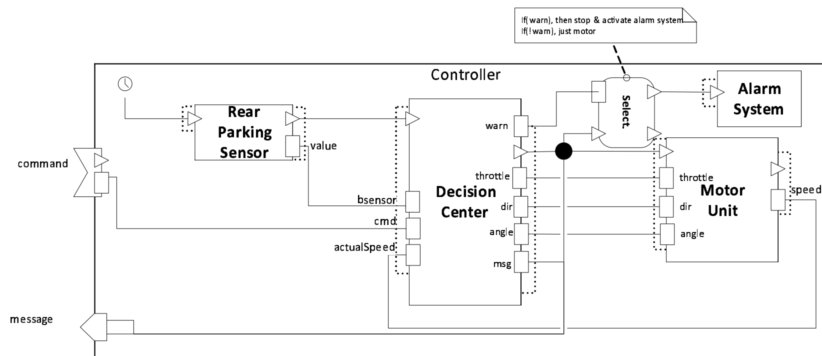


Figure 8.14: Controller System built out of ProSave components

The controller system works as follows: At each activation period, determined by a clock, the rear parking sensor is activated. Upon completion of the activity of the rear parking sensor, the decision center is next triggered. Basing its decision from the received instructions from the user (command input port), the actual speed and the data from the rear sensor, the throttle, direction and angle are outputted together with a message to be sent to the display. If the decision center signals a warn (i.e. the rear parking sensor has detected a collision), the alarm system is activated together with the motor. Otherwise, only the motor is activated.

The role of the *rear parking sensor* is, upon activation, to get inputs from the physical sensor and write into the output port whether the back sensor is activated or not.

The *Decision center* is the heart of the controller. It is in charge of dispatching information towards to appropriate subsystems. In the first iteration, it only consists of the manual regulator component as illustrated in Figure 8.15. The role of the *manual regulator* is simply to calculate the throttle and direction to apply according to the user inputs. If the vehicle is going backward, the alarm signals might get activated.

The Decision Center can be enhanced with an *Alarm Analyzer* component (see Figure 8.16). The Alarm Analyzer component is in charge of filtering the input from the sensors to eliminate erroneous signal. For instance, signals emanating from the back sensor should only happen when the vehicle is backing.

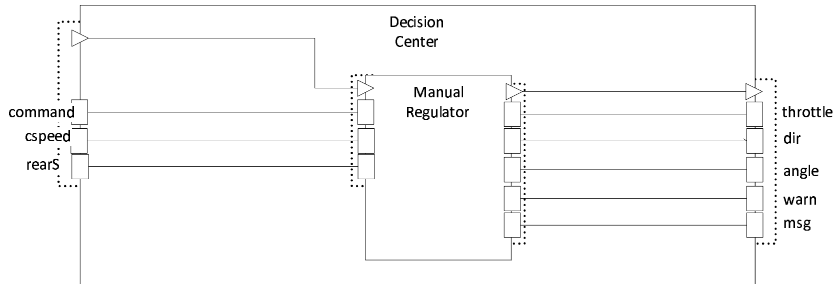


Figure 8.15: ProSave model of the Decision Center component

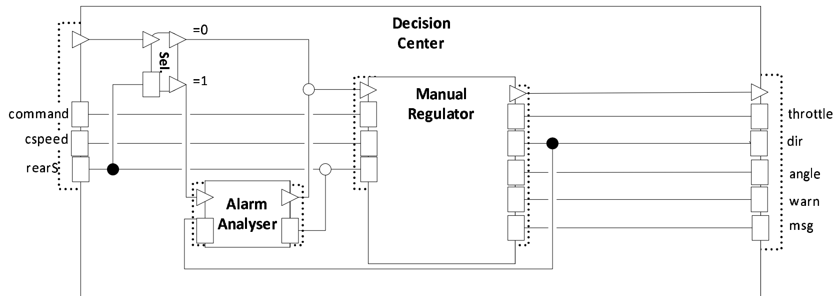


Figure 8.16: ProSave model of the Decision Center component with the Alarm Analyzer component

Analysing the components and the system

From the sole design of the Controller component described in the previous section, several analyses can be performed. Similar process can be repeated for any component present in the system, including the system itself. The purpose of these analyses is to assess the feasibility of the design and, if needed, investigate alternative solutions early in the development process. For instance, one may want to evaluate whether the advanced decision center can be used in place of the simpler version. In using expert estimates on the sub-components, extra-functional properties values such as timing properties can be derived for the composite component. This is possible thanks to the restrictive semantics of the ProCom component model. In applying the timing analysis proposed by Carlson [91], newly derived values can be obtained and compared to the requirements set in the previous development stage. Table 8.11 show an excerpt of these values. Other types of analyses can also be applied.

Once, the remaining components have been implemented, each of them can be validated and verified with respect to functional and extra-functional properties using various analysis techniques. The values obtained can then be used to refine the estimations previously derived for the controller component. Once, all the missing components have been implemented, the system can be synthesized. New values can then be inserted from the System components that would correspond to its execution or simulation. Table 8.12 provides an excerpt of final values for the WCET.

8.3.4 Enhancing the Drive-By-Wire System with an Automatic Driving Functionality (Iteration 2)

In this section, we describe interesting aspects of the development process to enhance the drive-by-wire system from iteration 1 with an automatic-driving functionality. This corresponds to iteration 2 from Section 8.3.1, in which a vehicle should automatically trail a preceding vehicle or be driven manually. On the overall, the development process followed here is similar to the one in iteration 1, but with the difference that components developed in iteration 1 are reused. The purpose is here to illustrate how ProCom components and multi-value context-aware extra-functional properties are used in a context of reuse.

Table 8.11: Excerpt of WCET attribute instances for the Controller component and its sub-components

Architectural Element	**Attribute Values**
Controller	[-] 150 000 – Source: Requirement – Type: Estimation – Platform: RCX i1 – Validity Conditions: Platform=''RCX i1'' [-] 110 000 – Source: Early Analysis – Type: Estimation – Validity Conditions: ''Controller Figure 8.14 with simple decision center component'' [-] 120 000 – Source: Early Analysis – Type: Estimation – Validity Conditions: ''Controller Figure 8.14 with advanced decision center component''
Simple Decision Center	[-] 25 000 – Source: Estimation – Type: Estimation
Advanced Decision Center	[-] 35 000 – Source: Estimation – Type: Estimation
...	...

Table 8.12: Excerpt of the WCET attribute instances.

Architectural Element	Attribute Values
Controller	[-] 150 000 – Source: Requirement – Type: Estimation – Platform: RCX i1 – Validity: Platform=''RCX i1'' [-] ~~110 000~~ 139 663 – Source: ~~Early Analysis~~ Derived – Type: Estimation [-] 123 780 – Source: Measurement – Type: Estimation
Decision Center	[-] 28 150 – Source: Analysis with Bound-T – Type: Estimation – Analysis Output: ./models/BoundT/wcet.txt – Comment: all loops repeat 2 times – Platform: RCX i0 – Validity: Platform=RCX i1
Motor	[-] 18 888 – Source: Measurement – Type: Estimation – Platform: ''RCX i1'' – Comment: Based on 100 executions. – Validity: Platform=''RCX i1''
RPS	[-] 79 000 – Source: Analysis with Bound-T – Type: Guarantee – Analysis Output: ./models/BoundT/wcet.txt – Platform: RCX i0 – Validity: Platform=RCX i1
Alarm System	[-] 13 625 – Source: Analysis with Bound-T – Type: Guarantee – Analysis Output: ./models/BoundT/wcet.txt – Platform: ''RCX i1'' – Validity: Platform=''RCX i1''
...	...

Reusing from the Drive-by-Wire System

The first step of the development is here to identify the parts that must be adapted from the drive-by-wire system to provide the advanced functionality of the automatic driving. This requires to decide which components are needed, which ones should be removed or modified to map the new requirements for the system. The same must be done with the extra-functional properties for the components remaining in the design and the ones that are planned to be adapted.

In order to do this, we start from the system description for iteration 2 in Section 8.3.1. The principal differences with iteration 1 are in the ability of the system to estimate the distance to the preceding vehicle through a light sensor, decide on the action to do to follow it and hand back the control to the human driver.

Accordingly, the development process starts this time by reusing the system built in iteration 1, i.e. the ProSys "System" component. In reusing this component, several extra-functional properties are also made available in the same time since the two systems (the one from iteration 1 and the one from iteration 2) targets a similar platform with the same CPU. These extra-functional properties correspond to the artefacts produced or needed during iteration 1 such as use-cases, statechart, requirements, analysis values and measurements. If the content of the System component is unchanged, the values can be reused directly. However, if the content is modified, it is necessary to identify and use only suitable values.

In the case in which, the hardware platform for the system being built is different than the platform from the previous project, it is important to also identify the extra-functional properties which are platform independent and can be directly reused from the platform dependent values that will typically not be reused apart for being used as rough estimations. In that case, the validity conditions play an important role to identify the values that can be reused.

For example, the UML use-case diagram available in the System component does not match the functional requirements for the system any longer. To better understand the system, we decide to keep this diagram and adapt it to fit the new functional requirements. As a result, a new use-case called "Automatic Drive" is added to the original use-case (see Figure 8.17). This changes in the original use-case implies that the attribute instance UML use-case can be reused.

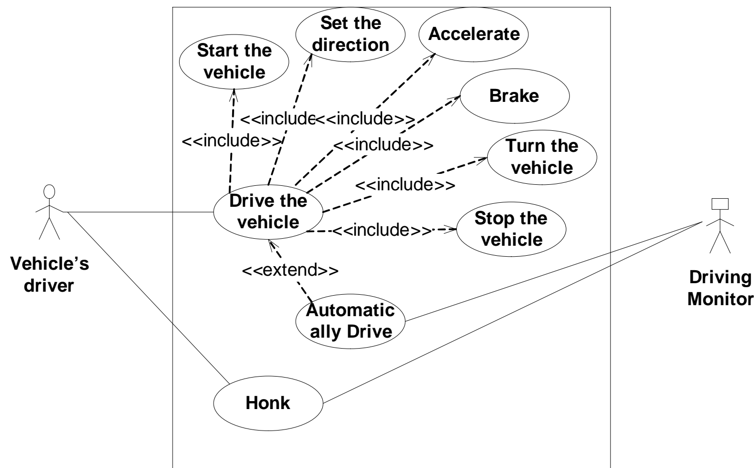


Figure 8.17: Use-case of the system for the ADS system.

Similarly, the extra-functional requirements M1 and M2 set for the drive-by-wire system are still valid in iteration 2 so they can be used as they are. New requirements can also be derived from the system description such as the following safety requirement:

> **ST2** The vehicle must maintain a distance to the target vehicle at least equals to twice the braking distance.

Then, the final architecture of the drive-by-wire system, i.e. the internal view of the System component, is examined to locate the changes that must be performed on the system. This corresponds to the solution shown in Figure 8.13 with the motor unit and the alarm system inside the controller component as illustrated in Figure 8.15. However, due to the additional requirements, none of the ProSys components used within the System component can be reused without modification. The Communication and HMI components require to be enhanced with an extra command to enable the activation of the ADS functionality. The Controller component also needs to be modified to cope with the new functional requirements. This implies that the attribute instances available for the System component cannot be reused directly. Yet, it is possible to reuse some of them as a conscious decision to provide some early estimations on the design. For instance, a physical size attribute instance mea-

sured for the HMI component can be used as a possible approximation in place of a measurement as shown in Table 8.13.

The Controller component also needs to be modified with the following changes:

- Addition of a distance sensor component that evaluates the distance to the preceding vehicle;
- Modification of the decision center to support the autonomous driving use-case.

From the components used to build the Controller component, the rear parking sensor, the Motor Unit and Alarm System ProSave components can be reused without modification together with their extra-functional properties. As for the use-case diagram of the System component, the statechart diagram within the controller component is adapted to support the autonomous driving state (see Figure 8.18). Table 8.13 shows some of attribute instances that can be reused in iteration 2.

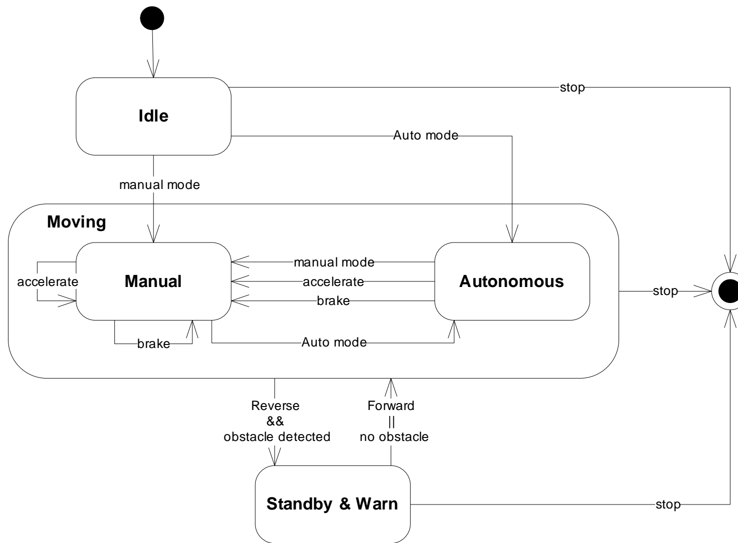


Figure 8.18: UML statechart diagram for the ADS system.

Table 8.13: Excerpt of Attribute Instances reused in Iteration 2.

Architectural Element	TypeID	Attribute Values
System	UML Use case	[-] ./models/UML/uc.uml
	Static Memory Usage	[-] 32 – Source: Requirement – Comment: M2
	Physical Size	[-] 10 – Source: Requirement – Comment: M1
HMI	Physical Size	[-] 2 – Source: Requirement – Comment: M1 [-] ~~1~~ 2 – Source: ~~Measurement~~ Estimation – CFlags: -O2 -Wall -fno-builtin -fomit-frame-pointer – **Comment: Estimation based on a previous project.**
Alarm System	Static Memory Usage	[-] 8 – Source: Analysis with Bound-T – Type: Guarantee – Analysis Output: ./models/BoundT/sMem.txt
	WCET	[-] 13 625 – Source: Analysis with Bound-T – Type: Guarantee – Analysis Output: ./models/BoundT/wcet.txt
Rear Parking Sensor	WCET	[-] 79 000 – Source: Analysis with Bound-T – Type: Guarantee – Analysis Output: ./models/BoundT/wcet.txt – Platform: RCX i1 – Validity: Platform=''RCX i1''
...	...	...

Architecting the Controller for the Autonomous Driving System

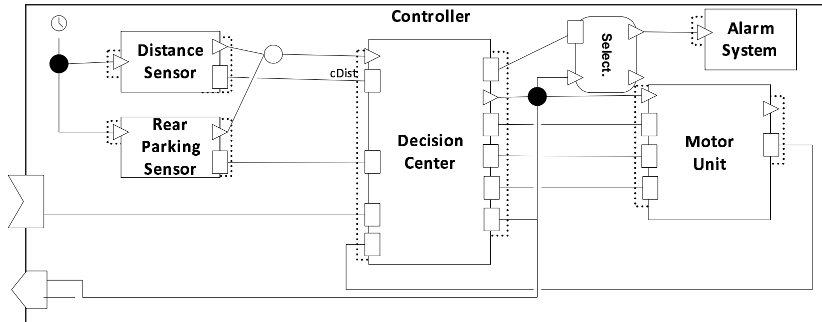


Figure 8.19: The ProSys Controller component for the ADS built out ProSave components.

First, in order for the system to be able to evaluate the distance to the preceding vehicle, it is necessary to add a distance sensor component to the design of the controller. As depicted in Figure 8.20, the distance sensor component is emulated through a light sensor that detects the amount of light reflected from the target vehicle and a distance estimator that uses the received amount of light to make a rough estimation of the distance between the two vehicles. The internal view of the new controller component is shown in Figure 8.19.

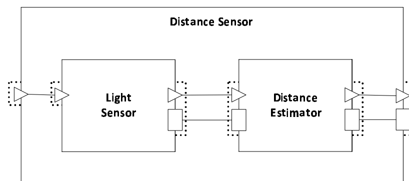


Figure 8.20: ProSave realization of the distance sensor

Further, the decision center component needs also to be modified to support the new functional requirements. As illustrated in Figure 8.21, two additional components are added to the decision center in the second iteration: the "*Autonomous Regulator*" and the "*Autonomous Target Finder*". The autonomous regulator is in charge of calculating the required throttle, direction and angle

with regards to the estimated distance to the target in order to maintain the appropriate distance to the target. In addition, if the rear parking signal is engaged and the car is backing, the autonomous regulator must signal a warning. It is assumed that the target to follow goes back and forth only and do not turn. If the target vehicle turns, the target will be lost. In that case, the autonomous target finder is in charge of relocating the target. Its role is to command the vehicle to progressively rotate the vehicle with increasing angle to try to find the target again. If after a certain time, the target has not been found, the Autonomous Target Finder raises an alarm, stops the vehicle that goes back in idle mode, waiting for new instructions. The Manual Regulator is reused from iteration 1.

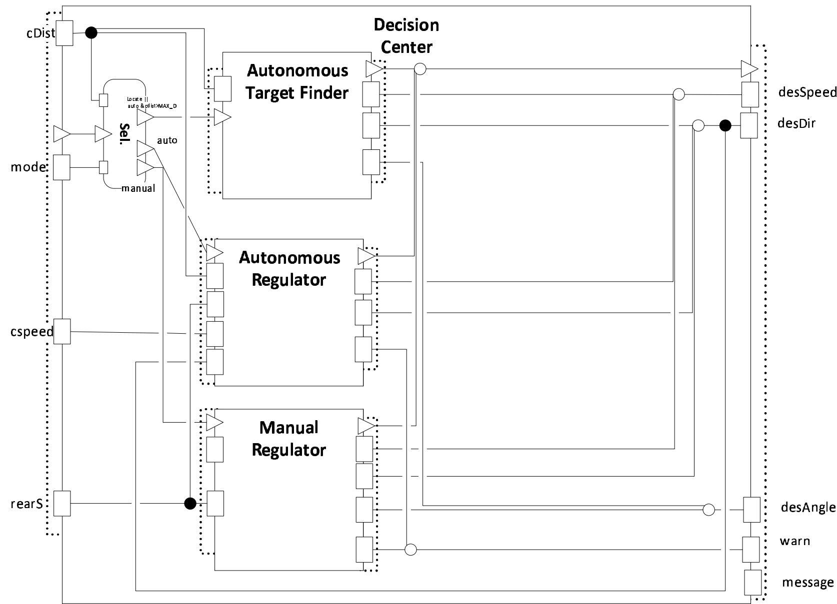


Figure 8.21: ProSave model of the Decision Center component in the second iteration

Similarly as for the previous iteration, the remaining component are then implemented and individually tested against functional and extra-functional properties before the final system being synthesised. New extra-functional property values can then be added to each component or to the system.

8.4 Summary

In Section 8.1, we have illustrated on the turntable example how a ProCom design and a dedicated analysis model can be used conjointly to perform early analysis of a system. Further, we have also shown how nLight can be used to packaged analysis artefacts and extra-functional property values into components. Through the realisation of this example, several outcomes can be discussed.

First, the integration of REMES into ProCom and the packaging as component was rather straightforward from the component-based design's point of view. Indeed, no change was required except simply registering a new attribute type into nLight. On the other hand, from the analysis point of view, more work was needed as it was necessary to adapt REMES to correspond to the semantics of ProCom in order to provide valuable analysis results.

In working on this example, it appears that it is quite difficult to determine which extra-functional properties would be good candidates to use as attributes. From the rich REMES model of a component, it is possible to express how e.g., the resource usage changes over time or in response to arriving messages, or how consumption of different resources are related. Accordingly, several isolated extra-functional properties can be extracted to be stored as separate attributes. For example, from the REMES model of Controller, a bound on the consumed energy can be extracted. Albeit very simple compared to the full REMES model, the Maximum Energy Consumption attribute instance attached to the Controller component would provide valuable information about the component and could serve as input to other analysis techniques. Selections of appropriate attribute candidates could be facilitated by having a precise quality process and development guidelines. Further, in extracting properties from a model a new challenge arises: how to ensure the consistency between the model and the extracted value?

With the personal navigation assistant system in Section 8.2, we have exemplified how inheritance policies are used in practice. The study of the system description highlighted the existence of dependencies between values of extra-functional properties. Such a dependency occurs within the scope of a project such as between the acquisition time of an antenna digital receiver and the response time of the parallel receiver. Another type of dependency which scope is global also exists as for instance between execution time, worst-case execution time and best-case execution. As a result, mechanisms to handle these dependencies should be provided.

With the last example of the automatic driving system in Section 8.3, we have demonstrated the use of the multi-valued context-aware extra-functional properties during the development of a system. The system has been developed using the ProCom component model and different analysis techniques to assess the extra-functional properties: component-based timing analysis based on ProCom semantics [91], static analysis with Bound-T [92] and measurements on the target platform. From this example, it has emerged that several important aspect must be considered for an efficient management of extra-functional properties in the development process. First, to foster the use of extra-functional properties in the development process, it is important that the various analysis techniques are tightly integrated with nLight. Otherwise, the analyses are performed outside the framework and this necessitates manual intervention to set the results as extra-functional properties. Furthermore, it is necessary to have clear rules to decide whether a new attribute value is a refinement of a previous value or a different value. Having this set of rules is particularly important for automated processes.

Chapter 9

Related Work

In this chapter, we relate the contributions presented in this thesis, namely a classification framework for component models, a framework to manage extra-functional properties, a new component model for distributed embedded systems and two integrated development environments to similar relevant work.

9.1 On Component Model Classification Frameworks

Over the last decade, several attempts have been made to identify key features of aspects of component software approaches: classification studies of components and interfaces ([93], [94]), interfaces, extra-functional properties ([28]), ADLs ([49]), component models ([50]), and characteristics of component models for particular business domains ([63]), among others.

The work presented by Yacoub in [93] and [94] does not consider any component model but rather focuses on practical issues of component utilisation and reutilization. In [93], the interface classification is split into two categories: application interfaces and platform interfaces. Application interfaces describe the information about the interaction with other components (messages protocol, timing issues to requests) whereas the platform aspect concentrates on the interaction between components and the executing platform. Similarly in [94] a model for characterizing components is proposed which reuses the classification model of interfaces from [93], where: a component is regarded as the description of three main items (informal description, externals and internals)

each of them split into several subelements. The informal description is connected with a set of features that relates to the use of a component in a team and over time. These features can influence the selection of a component such as: its age, its provenance, its level of reuse, its context, its intent and if there is any related component solving a similar problem. The externals are concerned with interaction mechanisms both with other application artefacts and with the platform (application interfaces, platform interfaces, role, integration phase, integration frameworks, technology and non-functional features). Finally the internals are concerned with elements related to the potential information needed during the development process of a system (nature, granularity, encapsulation, structural aspects, behavioural aspects, accessibility to source code).

A classification that is similar in spirit to our work, is proposed in [95]. This classification framework attempts to determine the core features of a software component. However, it differs from ours in including the identification of a component by a set of characteristics (unit of composition, reuse, interface, interoperability, granularity, hierarchy, visibility, composition, state, extensibility, marketability, and support for OO). The classification includes only business components and business solutions. One of the problems with this classification is the non-orthogonality of some of the characterized items.

In [49], where ADLs are classified, components are defined as basic elements of ADLs. The components are distinguished by the following features: interface, types, semantics, constraints, evolution, and non-functional properties.

In [63], a classification model is proposed to structure the CBSE body of knowledge. All research results are characterized according to several aspects (concepts, processes, roles, product concerns and business concerns, technology, off-the-shelf components and related development paradigms). Here, the component model is only considered as one of the fifty elements among the CBSE items. However, in this work, a more precise taxonomy of application domains is proposed. The paper identifies the following application domains in which component-based approaches are utilized: avionics, command and control, embedded systems, electronic commerce, finance, healthcare, real-time, simulation, telecommunications and, utilities.

In [7], several component models (JB, COM, MTS, CCM, .NET and OSGI) are mainly described according to the following criteria: interfaces and assembly using ACME notation, implementation, and lifecycle. The models are not compared or evaluated, but rather these characteristics are described for each component model.

In [50], a study of several component models is presented that considers the following aspects: syntax, semantics and composition through an idealized component-based development lifecycle. A smaller number of component models are considered (also UML and ADLs are included). Based on this study, a taxonomy centered on the composition criterion is proposed, which clarifies at which steps of the development process of a given component model, components can be composed and whether they can be retrieved from a repository to be composed. Furthermore, the different types of bindings (compositions) of some of the component models are discussed in more detail. This taxonomy does not consider extra-functional properties.

In comparison with all these works, the classification framework proposed in the thesis specifically focuses on component models and their intrinsic characteristics. These have been identified through a thorough and systematic literature review and analysis. However, the literature review could have been made more systematic in following the general guidelines proposed by Kitchenham in [96].

9.2 On Extra-Functional Properties

Extra-functional properties have gradually gained importance in software engineering to be viewed today as an absolute counterpart to functional properties. However, due to their complex nature, there is still no consensus on their definition, and on how they should be specified, used and assessed during the development. This results in a lack of support and the fact that they are seldom consider in practice. The same is also true in component-based software development as pointed out in [97] and [98].

Many works concerned with extra-functional properties can be found today in the literature. We use the categories below to group these works and some of them will be described and related to the thesis contributions in the following Sections 9.2.1, 9.2.2 and 9.2.3.

- *Contract-Oriented Approaches*: The works gathered in this category aims at proposing approaches to define contracts, and usage profiles which guaranty the correct extra-functional behaviour of the system at run-time. These approaches are often complemented with monitoring supports and negotiation policies. Hence, these approaches generally cover mainly two development phases: modelling to specify the extra-functional contract and runtime for the monitoring and possibly negociations of the properties.

- *Prediction-Oriented Approaches*: Approaches in this category aim at analysing extra-functional aspects with the intention of determining early in the development process whether a system will meet its extra-functional requirements or not. To a certain extend, these approaches contribute to the modelling activity of the system. These works can be further sub-categorised between "general-purpose" and "analysis-specific" approaches. These approaches can often be used conjointly with model-driven engineering techniques to generate implementation code complying with the envisaged specification and system model.

- *Fact-Oriented Approaches*: Approaches belonging to this category are intended to provide support during the development process to capture information about extra-functional properties on the system being developed in order to see whether the requirements are being satisfied. The approach proposed in this thesis falls into this category.

It is worth noting that works with different focuses on extra-functional properties in comparison to ones from the above categories can also be found in the body-of-literature. These works provide useful complementary information for the thesis contributions and can be grouped in the following categories:

- *Classification-Oriented Approaches*: Works belonging to this category are concerned with the identification, characterization and definition of a general structure to sort identified properties according to key characteristics. Some of classifications are generic [99, 100, 66, 101, 102, 103], whereas others focus on a specific category of properties such as dependability [1] or worst-case execution time [65]. Another subset of works from this category relates to specific property aspects such as composability [28].

  These works present the knowledge domain for extra-functional properties and in summarizing in a succinct form the key aspects of extra-functional properties, they can be used by approaches from other categories to know what must be considered. However, most of the proposed classifications are often non-orthogonal and non-consistent between each other. For instance, Laprie's dependability classification [1] collides with the standard IS09126 [66]. As a result, it is still difficult today to have a clear picture on how extra-functional properties should be represented, assessed and what factors influenced them.

- *Requirement-Orient Approaches*: Works on extra-functional requirements represents another facet of extra-functional properties, which many works such as [104, 105, 106, 107] are concerned with. This categories covers approaches to identify, express and manage extra-functional requirements.

9.2.1 Contract-Oriented Approaches

QML

QML [108, 109], Quality of service Modeling Language, proposes a general language for the specification of extra-functional properties. However, unlike our approach that considers extra-functional properties as stand-alone entities, QML envisages them within the scope of a contract (similar to our concept of category). In QML, a contract is specified through a set of extra-functional property specification called "dimensions". Dimensions are declared as follows: "$name : order\, data_format[unit]$" with, *order* describing the way the values are ordered (increasing or decreasing) and *data_format* corresponding to one of three proposed domain numeric, enumerated or set. The unit parameter is optional. When a contract is instantiated, optionally the value of each dimension can be constrained with authorized value range.

The primary motivation for QML is to support the creation of non-functional contracts for the development of distributed object-oriented systems in order to monitor whether extra-functional requirements are satisfied at runtime and dynamically adapt the system in case they are not. In QML, "attributable" elements are called profile and are limited to interface definition only but enables nonetheless to attached contracts for any entity of the interface definition as well. This includes operations and parameters for example. Refinement is also considered in the sense that a contract is a refinement of another contract if its dimensions are more constrained. QML enables properties monitoring, negotiating and conformance checking.

However, in comparison to our approach, the expressivity of QML is more limited and more informal. For example, a same property can be specified differently in two different contracts. Also, QML only offers a limited set of data type limited to numeric, set or enumeration values. This excludes the possibility to express properties through formula, distribution, etc.

QoSCL

Similarly to QML, QoSCL [110], the Quality of Service Constraint Language, also aims at the specification of extra-functional property contract. This language reuses the core concepts from QML and enriched them with two complementary aspects: *i)* dependencies between extra-functional properties can be defined ("*QoSCL allows a designer to declare a set of extra-functional properties, bonded between them into a network of relations (numerical constraints, formula, or empirical rules)*") and *ii)* techniques to automatically generate code for runtime monitoring and validate contract for component assembly are provided. However, QoSCL do not alleviate the drawbacks of QML mentioned above.

CQML

CQML [111], is a Component Quality Modelling Language built around three core concepts: *QoS characteristics* to define types of extra-functional properties, *QoS statements* to specify constraints on the values of the Qos characteristics and *QoS Profiles* to assign properties to components and parts of components. Further, QoS characteristics can also be grouped into categories. QoS Contract can be derived by agreement between QoS Profiles attached to different elements.

In CQML, types for extra-functional properties are declared through a unique name used as identifier and a value domain. Additionally, restrictions on the value domain, a value order and locally defined units can be used to complement the declaration. Extra-functional properties can also be specified with a statistical definition including for example its mean, variance, maximum value, minimum value, standard deviation, etc. However, how the statistical values have been obtained in not integrated into the language.

Refinements between extra-functional property types are envisaged through a specialisation paradigm enabling to restrict previously defined types. This is similar to the refinement concept proposed by QML for contract. Furthermore, CQML also considers component compositions in that sense that a QoS characteristic of a composite component can be derived from the same QoS characteristic of the sub-components. However, CQML does not take into consideration the dependencies that exist for example with the usage context, the resources, etc. as identified in [28] and [112].

Just as for QML, CQML does not provide a general-purpose type system and considering units as only informative. Value domains are limited to three predefined "type" consisting of numbers, set and enumerations only. CQML primary purpose is to provide a set of suitable concepts to support of extra-functional property aspects during development. CQML does not per se propose any monitoring, adaptation or agreement mechanisms. However, in [111] a QoS Framework including a QoS Monitor, an Adaptation Manager and a QoS Negotiator is described.

In [113], Röttger and Zschaler identified some limitations of QML and accordingly propose several enhancements for it. In particular, they insist on the need of having the computation model explicit, i.e. specifying what elements of that computational model can be enriched with extra-functional properties information. Furthermore, they propose to extend the language to consider resource-related properties such as CPU usage, memory.

9.2.2 Prediction-Oriented Approaches

The approaches described in this section use models enriched with extra-functional specific information for quantitative analysis purpose. These approaches can be separated into two groups: *general* that aims at providing ground for expressing a large variety of extra-functional properties in order to support their analysis, and *specific* that aims at a given type of analysis primarily. The specific prediction-oriented approaches are generally based on the particular needs of the targeting analysis to derive the necessary information which the models must be annotated with. The approaches described below consider component-based development. Other approaches aiming at the same thing include for example model-checking approaches such as UPPAAL.

UML Profiles

Several UML profiles have been proposed to provide for the lack of extra-functional property specification support in UML. Some of these profiles are UML SPT (the UML Profile for Schedulability, Performance and Time) [114], UML QoS & FT (the UML profile for Quality of Service and Fault Tolerance) [115], and MARTE (the UML profile for Modeling and Analysis of Real-Time and Embedded systems) [71]. These profiles are general prediction-oriented approaches in that sense that their main purpose is to provide modelling of extra-functional properties in order to specify and in some cases to enable analysis and validation of systems in early development phases. These

profiles allow to enrich any feature of UML diagrams with extra-functional properties annotations.

However, UML SPT and UML QoS & FT have identified shortcomings. UML SPT lacks expressivity and flexibility. It essentially supports timing properties with corresponding schedulability an performance analyses only. Furthermore, the profile cannot be extended to comply with specific user's needs. In comparison, UML QoS & FT covers more properties which can be fixed or dynamically managed. It also provides support for defining categories and a QoS catalog, i.e. a repository of extra-functional property specifications. But the language is bulkier while, alike UML SPT, it still lacks formal semantic. These limitations have led both profiles to be superseded by MARTE.

MARTE defines a general framework for the specification and design of extra-functional properties with the main intention of supporting any kind of analysis based on these specifications. To this end, the profile is structured in four packages of which the foundations one has a sub-package that specifically addresses extra-functional property modelling aspects: the NFP profile. The NFP profile is concerned with the specification on how to declare, qualify, and apply semantically well-formed non-functional concerns in real-time embedded system development. Some aspects of extra-functional property management proposed in this thesis emanate from MARTE and its NFP profile, namely the need for qualifiers that provide additional information on extra-functional property values (e.g. source, statistical measure, precision, etc.), and the need for a dedicated framework. With regards to our approach to extra-functional property management, MARTE offers a flexible solution to integrate extra-functional property information without having to modify the underlying model. However, MARTE is tight to the UML modelling language and does not focus on implementation and reuse as our approach.

Palladio

The Palladio Component Model (PCM) [48, 116] is a component model targeting business information systems. It specifically aims at enabling the predictions of performance and reliability properties in early development phases. This includes extra-functional properties such as response time, throughput and resource utilisation. The approach to predictions in the PCM is largely model-driven oriented and thus centered around several supporting models used by identified actors during the development process. A noticeable feature of this approach lies in the possibility to enhance these models with specific extra-functional property information that can be parameterised according to envi-

ronment influences such input parameters, resource usage, the usage of required services. This information is specified through a stochastic expression language called StoEx. The language is based on mathematical foundations for describing random variables and their distributions. This means that alike our approach, the PCM approach also acknowledges the need to make explicit the dependencies towards the usage context of extra-functional properties.

PECTs

From the Carnegie Mellon Software Engineering Institute (SEI), the Prediction-Enabled Component Technologies, shorten as PECTs [16, 117, 118], have been developed to facilitate predictability of run-time properties such as performance, safety and security. This approach stresses the importance of providing suitable quality prediction based on sound analysis theories for component models. In that sense, a PECT is defined for a specific component model (Pin for example [32]) and integrates in it a set of supporting analyses through dedicated reasoning frameworks. A reasoning framework consists of an analytical interface per property and corresponding analysis theories and models. Examples of reasoning frameworks for the Pin component model include ComFoRT [119], a model checking reasoning framework to determine whether safety and reliability requirements are satisfied; Covert [120] to discover buffer overflows in C programs and Lambda-* [121] for predicting timing properties such as average or worst-case latency of tasks.

In comparison to our approach that does not require changes in the components, each Pin components must be enriched with a dedicated analysis interface to enable the use of the corresponding reasoning framework. However, the approach also intends at making explicit assumptions on environment and usage context such as the used scheduling policies, target platform, etc. In that respect, this ties up to our view that these must be made explicit and maintain for future reference. In PECT, properties are simply defined as an n-tuple $\langle name, value, \dots \rangle$, where the triple-dot punctuation mark represents an arbitrary number of property-specific parameters. The need to attach extra-functional properties to different entities is also recognized and properties can be attached to component, assembly, pin, reaction, environment, and environment service.

9.2.3 Fact-Oriented Approaches

Credentials

In [46], extra-functional properties are represented through the concept of credentials. A credential is defined as a triple $\langle Attribute, Value, Credibility \rangle$, for which *Attribute* identifies the component property, *Value* the corresponding data, and *Credibility* describes how the value has been obtained. Credentials are considered to be incremental and evolving specifications that requires attribute names to be registered. However, details on the concepts remain rather vague and succinct. For example, for the concept of credibility, an enumeration of possible values is simply given.

The concept of credentials has been integrated in Ensembles [122] and in SaveCCM [14]. In both approaches, credentials are attached to component only. An extension to Ensembles is proposed in [7] that allows credentials for interfaces and their operations. This concept has also inspired the first steps of the thesis work on extra-functional properties.

A Formal Specification Framework

In [123, 124], Zschaler proposes a formal specification framework for the specification of extra-functional properties in component-based systems. In many respects, this approach is closely related to the one presented in this thesis. First, a distinction between intrinsic and extrinsic property is introduced. An *intrinsic property* is a property for which the value is solely dependent upon the component's implementation. In contrast, an *extrinsic property* is a property which value is dependent upon the component usage, i.e. the value is influenced by factors outside the component. This dichotomy relates to the difference between attaching an attribute to a component type (intrinsic property) and attaching an attribute to an instance of the component type (extrinsic property). Second, the approach makes use of models to formally support extra-functional property specification. In particular, models called context models are used to analyse specific properties in order to obtain values. Likewise in our approach, we advocate that extra-functional property assessment can be facilitated by the use of a hierarchical component model with a precise semantics throughout the development process complemented by suitable allocation and platform model. Similarly, additional models can also be used to assess extra-functional property values as illustrated in Chapter 8. Third, the concept of measurements is introduced. A measurement describes how a value can be obtained. However, it is unclear how the measurement is linked to the property once obtained. The degree of formalism of the approach is high and extensive

as it relies on Temporal Logic of Actions (TLA+)[125] for the specification of all the concepts of the framework. However, this renders the approach rather unintuitive and due to the use of TLA+ some extra-functional properties cannot be expressed by the approach such as stochastic properties or properties such as learnability and maintainability.

Global/Local Repositories for Extra-Functional Properties

Some characteristics of the approach proposed by Ježek and Brada in [126, 127] have directly been inspired by the work on extra-functional properties presented in this thesis. These are namely the main objective of the approach and the necessity of having a repository for extra-functional properties. The approach shares our vision of a dedicated framework that enables to systematically specify, integrate, manage and assess extra-functional properties in component-based systems. Similarly, the framework is not tight to a specific component model but instead allows enriching any component model specification with a suitable support for extra-functional properties. Both approaches differ however in their coverage of component-based development process. Whereas our approach aims at supporting extra-functional properties in the whole development process from early design up to synthesis (i.e. excluding requirement and execution phases), Ježek and Brada's targets the component packaging and selection stages in which components are seen as black-box. As a consequence, extra-functional properties are only attached to component types and features thereof. Likewise, the approach also acknowledges the use of a repository of extra-functional properties. In contrast of using the repository as a catalog of available properties only, the approach uses a system of repositories distinguishing a *global repository* storing extra-functional property types from *local repositories* gathering extra-functional property values evaluated in a given environment. However, the specification does not state what aspects of the environment are embodied in a given local repository and how these aspects are related to the extra-functional property values. Similarly, the approach does not maintain information on how the value has been assessed and to which extend the value can be reused, i.e. is this value reusable in the same context? or if it can be used in a different environment. The concepts of metadata is also introduced informally as a record containing any additional information such as property unit and allowed distinctive name for the property. However, such a definition does not provide the flexibility of metadata concept introduced in Chapter 3 and even implies that a value with a different unit should be a different extra-functional property, hence weakening the semantics behind a property.

9.3 On Embedded System Development

9.3.1 Component Models

A broad range of component models exists nowadays, either general purpose or dedicated component models, as compiled in various classifications (as in [7] or [50] for instance). However, few component models actually target the development of embedded systems and most of them focus on a specific domain only. Using the component models from Chapter 2 as a basis, this section compare the component models targeting embedded systems with ProCom

In the automotive domain, the AUTOSAR (AUTomotive Open System ARchitecture) consortium [128] is the first large-scaled initiative to gather manufacturers, suppliers and tool developers from the automotive field to establish an open and standardised software architecture for the automotive domain enabling component-based software design modelling. Through this common standard, the vision of AUTOSAR is to facilitate the exchange of solutions (including software components) between different vehicle platforms and subsystem manufacturers as well as between vehicle product lines. In that sense, AUTOSAR targets the upper part of the granularity scale of the proposed conceptual component model. Similar to our approach, AUTOSAR relies upon the use of a component-based software design model. However, the two approaches have principal differences. In particular, AUTOSAR component model proposes both pipe and filter and client-server paradigms communicating transparently across the architecture through the use of standardised interfaces. Although targeting the development of applications for the automotive domain, the first versions of AUTOSAR were lacking support to express and analyse extra-functional properties in particular timing properties as for instance worst-case execution time or end-to-end deadline. AUTOSAR 4.0, done in cooperation with the TIMMO project [129] and EAST-ADL [130], intends to tackle this lack by an extension of the current metamodel. In particular, the TIMMO project intends to propose a standardised infrastructure to manage timing properties and enable their analysis at all abstraction levels from early design to deployment.

A second initiative that shows the growing interest from the automotive domain in component-based software development comes from Bosch with BlueArX [9, 131]. Also based on a design-time component model, BlueArX differentiates itself from AUTOSAR in supporting timing and other non-functional requirements as well as in focusing on complete development process for single ECUs. To this respect, BlueArX is relatively close to the objectives and contributions presented in this thesis in particular with regards to the lower layer of the component model (ProSave). However, differences ex-

ist. First, through the ProSys layer of the component model, ProCom intends to support also the development of embedded software systems distributed across several ECUs. Another difference lays in the proposed support to integrate analysis. Whereas extra-functional properties can be associated with any entities of the ProCom component model (components, ports, services, connections or component instances) through the attribute framework extension, BlueArX on the other hand endows components with an additional analytical interface to perform analysis either at system- or component-scale. In a recent work [10], BlueArX has been extended to support the analysis of timing properties in relation to operational mode, a feature which is not supported yet within ProCom.

Developed in a close cooperation between Arcticus Systems AB and Mälardalen University, the Rubus Component Technology [11] is another example of an industrial use of component-based approach in the vehicular domain. Similarly to ProCom, the RUBUS component model focuses on expressivity and analysability through a restrictive component model. However, the Rubus component model allows the specification of timing properties only and is not primarily concerned with reuse.

The contributions found in this thesis are largely inspired by previous work done at Mälardalen University on the elaboration of a component model for vehicular domain. SaveCCM [76] is a design-time component model consisting of a few design entities with a restrictive "Read-Execute-Write" execution semantics and communicating through a "pipe & filter" paradigm in which the control- and data-flows are distinctly separated. Having such a restrictive semantics, it enables formal validation and verification of the system already in early phase of the development process, prior any implementation as well as automated part of the transformations into an executable system as explained in [24]. ProCom is built on the knowledge and experiment gained from the development of SaveCCM and tries to alleviate some of the restrictions and drawbacks of SaveCCM in particular in strengthening the concept of components, considering distribution and handling functional and extra-functional properties in a more systematic way. Whereas the ProSave layer is to a large extent directly inspired from SaveCCM, the upper layer (ProSys) aims at addressing the distribution of subsystems, which was not addressed within SaveCCM.

In the field of consumer electronics, Philips has developed and successfully used the Koala component model [12] for the production of various consumer electronic product families (TV, DVD, etc.). In comparison to the aforementioned initiatives, Koala is less oriented towards safety-critical applications than what exists in the automotive domain for example. However, as Koala

still targets severely constrained embedded systems, it pays a special attention to static resource usage, such as static memory for instance, but it lacks support for managing other extra-functional properties. The dependencies between properties are handled through diversity spreadsheet, which is a mechanism outside the component. Koala has served as input in the Robocop [15] project done in collaboration between Philips and Eindhoven Technical University. Similarly to ProCom, Robocop considers components as a collection of models covering the different aspects of the development process. Models are also used to manage extra-functional properties as for instance the resource model, which describes the resource consumption of components in terms of mathematical cost functions, or the behavioural model, which specifies the sequence in which the operations of the component must be invoked. Additional models can be created.

Pecos [132] is a joined project between ABB Corporate Research and Bern University. Its goal is to provide an environment that supports specification, composition, configuration checking and deployment for a specific type of reactive embedded systems (field devices) built from software components. Contrary to ProCom for which the components of each layer have their own execution semantics, i.e. ProSys components are active whereas ProSave components are passive, the two types are put together in Pecos. Also, since components in Pecos have only data ports, there is a need for an additional type of component, called event component, which activation is triggered by the arrival of an event. With regards to extra-functional properties, Pecos enables the specification in a name-value pair format in order to investigate the prediction of the timing and memory usage of embedded systems. However, this specification is limited to name-value pairs in difference to the possibility offered to specify extra-functional properties in ProCom.

Pin [32], a component model developed at Carnegie Mellon Software Engineering Institute (SEI), serves as basis for the prediction-enabled component technologies (PECTs) which aims at attaining predictability of run-time properties such as performance, safety and security. Alike our approach, PECT stresses the importance of providing suitable quality prediction based on analysis theories. However, the methods to integrate analysis differ. Whereas ProCom relies on an external attribute framework as means to handle functional and extra-functional properties resulting from different analysis techniques, PECT is centered around a reasoning framework consisting of analytical interfaces used to specify specific properties, and corresponding analysis theories to enable the prediction of these properties. Also in comparison to ProCom, Pin is a flat component model which does not support distribution.

9.3.2 Alternative Approaches

This section correlates our work with other approaches that are not primarily concerned with the principles and methods advocated in CBSE but are still intended to support the development of distributed embedded systems.

In the automation domain, the standards IEC-61131 [40] and its successor IEC-61499 [55] proposed by the International Electrotechnical Commission are well-established technologies for the design of Programmable Logic Controllers. Whereas IEC-61131 allows to graphically compose systems out of function blocks, IEC-61499 has been developed to enforce encapsulation and provide a support for distribution. From a design perspective, ProCom shares some similarities with these graphical languages, in particular the encapsulated entities communicating with a “pipe & filter” paradigm with explicit separation between data- and control-flow, and the distribution support. However, the semantics associated with the function blocks are weaker compared to the ProCom components, and the standards lack support for specifying and managing extra-functional properties and their analysis. This holds back the possibility for formal analysis of the systems under development, which is one of the major objectives this thesis aims at.

In the automotive domain, alike ProCom, EAST-ADL (Electronic Architecture and Software Technology – Architecture Description Language) [130] aims at providing a support for the complete development of distributed embedded systems by taking into consideration the hardware, software and environment development assets. Although both approaches share similar objectives, they differ in the way those objectives are approached. Whereas ProCom emphases components as assets for capturing development information thus aiming at reusability, EAST-ADL focuses on architecture description to structure it. In EAST-ADL information is structured into five abstraction levels, which describe the functionalities from several standpoints. Each entity of a level realizes the entities of the higher abstraction levels. ProCom covers three of these levels (analysis level, design level and implementation level), and leaves out the electronic feature design (vehicle level) and the support for the deployment of the final binary (operational level). Similarly to ProCom, EAST-ADL also supports modelling of non-structural aspects such as behavioural description but covers in addition validation and verification activities as well as management of requirements. EAST-ADL was originally developed as an EAST-EEA ITEA project involving car manufacturers and suppliers and now it is refined as a part of ATESST project to be aligned with the major standardization efforts existing in the automotive and real-time domains (AUTOSAR, MARTE, and SysML).

The Architecture and Analysis Description Language (AADL) [133, 134], formerly known as Avionics Architecture Description Language, is a standardization effort led by the Society of Automotive Engineers (SAE) to provide support for the development of real-time and safety-critical embedded systems for aerospace, avionics, robotic and automotive domains. Consequently, AADL stresses the importance of analysis to meet the particular constraints and requirements of the envisaged target domains. It provides a formal hierarchical description of the systems including properties to support the use of various formal analysis techniques related to timing, resources, safety and reliability with the aim of validating, verifying and performing trade-off analysis of the system. Properties are defined as a triple (Name, Type, Value) that can be attached to different entities and can have specific instance values. To this respect, AADL is comparable to ProCom with nLight. However, in comparison to ProCom, AADL is "only" a description language and does not provide links to design and implementation technologies. In that sense, it decomposes the system in a top-down manner specifying entities and how they interact and are integrated together without providing any implementation details. Thus, AADL is not primarily concerned with reusability issues. On the other hand, AADL includes some features that could be interesting to take into consideration in the further development of ProCom such as the specification of execution platforms and operational modes.

Chapter 10

Conclusions and Future Work

This chapter concludes the thesis by taking a step back to the research questions introduced in Chapter 1 to put them in perspective with the thesis contributions discussed in the previous chapters. The chapter starts in Section 10.1 by summarising the novel contributions of the thesis before discussing, in Section 10.2, their relation and participation towards the overall research goal and the research questions. Finally in Section 10.3, the chapter ends by identifying and introducing possible directions in which the work could be continued.

10.1 Summary

In this thesis, we have investigated how extra-functional properties should be efficiently treated in component-based development of embedded systems. This investigation has led to four main contributions, namely *i*) a component model classification framework, *ii*) a general framework for specifying, integrating and managing extra-functional properties in component-based development, *iii*) a component model dedicated to embedded system development, and *iv*) two integrated development environments supporting the two previous contributions.

In Chapter 2, we have first examined the state-of-the-art and state-of-practice of component-based development through their keystone, that is the component model. As a result, twenty-four component models have been thoroughly studied, also with respect to how they specifically support extra-functional properties. From this study, a component model classification framework has been devised, which presents in a clear and concise manner the important features of component models. This facilitates identifying the common characteristics and principal differences between the considered component models. In particular, it puts in evidence the lack of suitable management support for extra-functional properties in component models.

Having identified this key fact, we have looked deeper into what challenges the use of extra-functional properties poses for component-based development. Based on this, we have formally defined, in Chapter 3, the concept of multi-valued context-aware extra-functional properties. This concept highlights two important aspects emanating from the use of extra-functional properties in component-based development: *i*) their multi-valued nature, that is, several extra-functional property values can be equally valid in a given development context and therefore must co-exist, and *ii*) their context-awareness, i.e. extra-functional property values typically depend on their usage context and this dependence must be captured and made explicit in order to facilitate reusing the extra-functional properties together with the component they describe.

Building upon this formal basis, support mechanisms have been identified in Chapter 4. These mechanisms were necessary to address the challenges introduced by the multi-valued and context-sensitivity nature of extra-functional properties, namely redundancy, applicability, and confidentiality. Furthermore, solutions to facilitate selection of extra-functional properties and refinement between component types and component instances have also been described. This has laid the first step towards the systematic consideration and use of extra-functional properties in component-based development. Another step derives from the realisation of nLight, a general framework for specifying, integrating and managing extra-functional properties in component-based development. The key implementation details of the framework have been described in Chapter 5.

In Chapter 6, we have investigated how a component model can be specified to facilitate the analysis of extra-functional properties while at the same time presenting the same concept of components throughout a development process starting from early modelling up to synthesis. The focus has been placed here specifically on component-based development for embedded systems due to the importance of ensuring extra-functional properties such as tim-

ing, resource usage, fault-tolerance in this particular domain. The study of this concern has led to the creation of a new component model dedicated to embedded systems, called ProCom. Its characteristic features, detailed in Chapter 6, include *i*) a two-layered structure to cover the different requirements that exist at different granularity levels of embedded systems, *ii*) a restricted execution semantics to facilitate analysis, and *iii*) a rich design-time component concept to package the various artefacts require or produced during the development process.

In Chapter 7, we have described the integrated development environments that have been built to support the development of embedded systems complying with the solutions proposed in the previous chapters.

10.2 Discussions

Research Question 1:

> *What characteristics of a component model facilitate software design of distributed embedded systems?*

Based on an analysis of the component model classification framework and an evaluation of the requirements for embedded system development, we have identified a number of characteristics that a component-based embedded system development and its associated component model should be endowed with. A component model should support:

- Different abstraction levels (i.e. the coexistence of components in an early design phase and fully realised components).
- The different concerns that exist at different granularity levels (i.e. an high-level view of loosely coupled complex subsystems together with a low-level view of small non-distributed functionalities similar to control loops).
- Platform awareness while still being platform independent.
- A seamless integration of various analysis techniques.
- A systematic management support for extra-functional properties.

In addition, as identified by Åkerholm in [135] for the development of hard-real time embedded control software, the component model semantics should be limited to support analysis of important extra-functional properties such as timing, safety or reliability. With regards to efficiency of software development of possibly less constrained embedded systems, this implies finding the appropriate trade-off between flexibility on one hand and analysability and predictability on the other hand. We approached this problem by alleviating some of the restrictions present in SaveCCM — in particular, for the ProSys level which requires more flexibility than ProSave since it deals with distributed active subsystems executing concurrently — while reinforcing the concept of components as a uniform notion throughout the development process. In spite of this, ProCom provides a semantic precise enough to be formally expressed through timed finite state machines as demonstrated in [73]. This permits various analysis tools to benefit from the features offered by ProCom to perform specific analysis already in early development phases, hence potentially improving the development process performance and costs. Several analysis techniques have been built and tested with the strict semantic proposed by ProCom: parametric worst-case execution time analysis [83], another timing analysis [91], model checking of behavioural models [74] and fault-propagation [85]. Additionally, this also allows to perform synthesis of systems fully compliant with the underlying semantic of the component model as described in [86].

The strong coupling between target platform specification and software implementation is an important challenge since the correctness of many analysis results and values of extra-functional properties strongly depend upon the target platform specification and the deployment configuration. Postponing the access to this information to a late development stage could result in incorrect design and implementation of the system to be executed, possibly leading to a costly redesign and re-implementation of the erroneous parts of the system. Such information is also foremost important for extra-functional properties. Yet, breaking the hardware abstraction and making the target specification part of the component model is not a suitable solution since this would makes all components platform dependant and hinder their reuse, breaking one fundament of CBSE. Such a problem arises for example when sensors and actuators are used in primitive components that are used in turn to create composite components. An appropriate solution lays somewhere in between those two extreme solutions. Solutions to tackle this problem are presented in [136] for the integration of sensors and actuators in component models and in [137] for component allocations.

Research Question 2:

> *What mechanisms are suitable to support the management of extra-functional properties within a component model?*

Answering this question corresponds to finding an appropriate way to specify, integrate and handle functional and extra-functional properties in a component model in a systematic way. Thus, after identifying the main challenges of dealing with extra-functional properties in component-based development, we have addressed this question through the concepts discussed in Chapters 3 and 4. We have formally defined the concept of multi-valued context-aware extra-functional properties and identified necessary supporting mechanisms to facilitate their seamless management. These concepts have been implemented in nLight. In summary, our approach to multi-valued context-aware extra-functional property management combines a formal definition for specifying extra-functional properties with techniques outside this specification, such as a property registry and property selection mechanisms, to ensure the correctness of their utilisation in the current development context.

The concept of multi-valued context-aware extra-functional properties makes explicit the multi-valued and context-sensitivity nature of extra-functional properties. In that sense that it enables *i*) handling simultaneously the specification of multiple values for a property, where each value is identified through the provision of suitable metadata and/or the context under which the value has been obtained; and *ii*) expressing the dependencies of the values towards outside parameters.

This approach can also be used to integrate the specification of functional properties without hampering the utilisation of interfaces. In this context, functional properties refer to the modelling of the behaviour of the components in a format suitable for analysis techniques such as timed automata model. By this, our intention is to increase the analysability and predictability of component-based embedded systems, and enabling a seamless and uniform integration of existing analysis and predictions theories into component models.

However, the concept of multi-valued context-aware extra-functional properties introduces complexity in the design process in several ways as identified in Chapter 4. In addition to the possibility to have multiple values assessed at different point of time or by different techniques, it also envisions delegating the declarations of needed properties to, for example, the developers of the analysis techniques who know best the types of information they need as input

and that they produce as outputs. In the end, this could result in an explosion of property definitions in the registry. A possible solution would be to rely on a standardized catalogue of properties similarly to what exists for units (SI), date and time representation (ISO 8601) or the standard for evaluation of software quality (ISO 9126). Another possibility would be to develop techniques to avoid the definition of equivalent properties, i.e. a property which semantics is strictly the same to a property already present in the repository but some parameters are slightly different. For example, "worst-case execution in ms" and "statistical worst-case execution time in ms" should not be two different extra-functional property types but only one: the "worst-case execution time". Its definition should correspond strictly to the semantics of property and to the parameters should be handled within the property with for example conversion mechanisms.

Our approach to integrate extra-functional properties in component models reveals a lot of information concerning the details of the implementation of the components. Although this is not a major issue for in-house development, it naturally becomes more problematic for its utilisation in the development of systems or components for which the implementation details must remain hidden such as COTS components since all the models that have served for analysis are packaged together with the components. A solution could be to provide mechanisms to identify and automatically remove confidential information when components are distributed to third parties.

Research Question 3:

> *How can the different aspects of component-based development for embedded systems be seamlessly integrated into a development environment?*

From the knowledge gained from the development and use of the Save-IDE and PRIDE, described in Chapter 7, we can identify several factors that contribute to facilitate the integration of different aspects of a component-based approach into a common environment. First of all, it is important to have the set of precisely specified theories and concepts that will be embodied within the IDE. This serves three main purpose:

1) defining the core concepts that will form the backbone of the IDE,
2) clarifying the relationships, dependencies and gaps between these concepts, and
3) identifying what should be the main features of the IDE.

In the case of the Save-IDE, we strictly followed the concepts and theories developed for the SaveCCT approach [14], namely *i*) a design-time component model, SaveCCM [76], with a strict execution semantics for the components, *ii*) early formal analysis of timing and behaviour properties [75], and *iii*) dedicated synthesis.

Similarly the objective of PRIDE has been to provide a flexible component-based development process from early design up to synthesis supported by an interlacing of various analysis techniques. The development has been centred on the following theories and concepts: *i*) the rich-design time component concept as the main development unit throughout the development process *ii*) the ProCom component model with its well-defined semantics, *iii*) a systematic management support for extra-functional properties [18], *iv*) the REMES model and its corresponding timing, behaviour and resources analyses [74, 84] , several timing analysis techniques [83, 91], and a fault-propagation analysis techniques [85], and *v*) a flexible synthesis that fully respect the semantics of the ProCom component model [86].

This implies that, despite its known influence on tool integration (either positive or negative), the format of a development artefact is not so important since translating from one format into another is generally possible. What is more important, on the other hand, is the information behind this artefact: its content and its purpose, i.e. what is this artefact about and how it is supposed to be used. Combining the core concepts together can be facilitated by using model-driven development techniques.

Several aspects of the development process are generally tightly related such a architectural design decision that depends upon results from a given analysis techniques. Accordingly, it is necessary to provide traceability between the different development artefacts involved in the development, and to track changes and possibly propagate them.

Furthermore, it is also valuable to provide flexible and extensible mechanisms to facilitate handling new requirements and the addition of new activities. In both, the Save-IDE and PRIDE, we support this needs through the use of the Eclipse. However, PRIDE, through nLight and the analysis framework provides additional means to seamlessly integrate new analysis techniques.

10.3 Future Work

In this section, we describe some of the directions in which the contributions presented in the thesis can be continued.

Industrial Validation of ProCom and nLight

The study examples presented in Chapter 8 show that different embedded systems can be modelled and implemented in ProCom. However, the validation of ProCom and nLight on a realistic industrial distributed embedded system remains to be done. Such an evaluation would allow assessing the strengths and limitations of the component model and the attribute framework as well as their practical impacts on the development process.

Relational Database for Managing Multi-Value Context-Aware Extra-Functional Properties

The current implementation based on the Eclipse Modelling Framework focuses on facilitating the integration of nLight with component models specified through a metamodel that defines their key concepts and their relationships. This implementation has proven useful in many ways, notably to automatically set specific elements of a component model as attributables. However, due to the highly heterogeneous and structured nature of multi-valued context-aware extra-functional properties, managing them in this way during a development process can become challenging. As the development process progresses, the complexity of the systems grow: many components with many extra-functional properties must be envisaged. This may lead to a situation in which the amount of information to look through is overwhelming as it is necessary to identify the values that are relevant in the current development context.

A possible solution to this problem lies in the development of a relational data model which is an acknowledged solution to handle huge amount of data with complex relationships between them. Additionally, some of the identified challenges for the management of multi-valued context-aware extra-functional properties might have already known solutions in the database domain. Indeed, databases propose solutions to data indexing, data retrieval and selection through queries as well as enabling to reduce data redundancy and facilitate data storage.

One of the challenges here is to merge the existing model-driven approach based on metamodelling, with an approach using embedded databases so that, ideally, the data management aspect of extra-functional properties remains hidden from the users of nLight.

Inter-Property Dependencies

When using extra-functional properties in practice within system development, as in Chapter 8 with the extended examples, dependencies between extra-functional properties have become visible: *global dependencies* between attribute types and *local dependencies* which are dependencies that only make sense in a particular project. A global dependency is a relation that always exists between values of two or more extra-functional properties. For example, if one considers the value of the execution time as an interval bounded by the best-case execution time as lower limit and worst-case execution as the upper one. Then, there exists a dependency between execution time, best-case execution time and worst-case execution time. This dependency must be considered and reflected also in nLight. On the other hand, a local dependency is a relation between extra-functional properties that emerges from the specific requirements and design choices of a system development. For example, in the study example of the GPS in Chapter 8, the response time of the GPS depends on the acquisition time of the receiver. The multi-valued nature of extra-functional properties poses here an interesting challenge with respect to the inter-property dependency problem.

Validity Conditions Language

As mentioned in Chapter 3, an important characteristic of multi-valued context-aware extra-functional properties lays in the concept of validity conditions that specify the criteria under which a value is known to be correct. These criteria are quite diverse, including for example usage context, specific analysis techniques, and target platforms. It is important that these criteria are somehow linked to the current development environment of the system against which they must be checked. As such, a validity condition language must be specified and it should enable the creation of mechanisms to verify whether a given extra-functional property value is valid in the current development context.

Multi-Valued Context-Aware Extra-Functional Property Comparison

As explained in Chapter 4, an important challenge for the systematic management of multi-valued context-aware extra-functional properties in component-based development is to enable conciseness. The first step towards this is to identify duplicates and redundant values, i.e. values that are equivalent. Due to the complex structure of multi-valued context-aware extra-functional properties, this is not straightforward. A clear equivalence between values can be established when all data, metadata and validity conditions are strictly the same. Yet, not all of these characteristics are always pertinent to determine equivalent values. For example, at a certain point in time in the development process, a developer or an analyst might not be interested in differentiating between the sources of extra-functional property values to perform early estimations. In that particular example, this implies that the values should be compared in disregarding the source metadata. Another case relates metadata and validity conditions. For instance, if a certain attribute value is declared as platform independent through the validity conditions, then again the platform metadata should be discarded from the comparison.

All those aspects pose an interesting challenge to solve. In particular, being able to determine which values are the same will enable to contribute to the conciseness principle in enabling, for instance, to merge strictly redundant values.

Bibliography

[1] Jean-Claude Laprie. Dependable Computing and Fault Tolerance : Concepts and Terminology. In *Fault-Tolerant Computing, 1995, 'Highlights from Twenty-Five Years', Twenty-Fifth International Symposium on*, pages 2+, 1995.

[2] Panagiotis Tsarchopoulos. European Research in Embedded Systems. In *Embedded Computer Systems: Architectures, Modeling, and Simulation, 6th International Workshop, SAMOS 2006, Samos, Greece, July 17-20, 2006, Proceedings*, pages 2–4, 2006.

[3] K. Venkatesh Prasad, Manfred Broy, and Ingolf Krueger. Scanning Advances in Aerospace & Automobile Software Technology. *Proceedings of the IEEE*, 98(4):510 –514, april 2010.

[4] Robert Bosch GmbH. CAN Specification, Version 2.0. Technical Report ISO 11898, 1991.

[5] LIN Consortium. LIN Protocol Specification, Revision 2, September 2003.
`http://www.lin-subbus.org/`.

[6] Manfred Broy. Challenges in Automotive Software Engineering. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 33–42. ACM, 2006.

[7] Ivica Crnkovic and Magnus Larsson. *Building Reliable Component-Based Software Systems*. Artech House, Inc., Norwood, MA, USA, 2002.

[8] AUTOSAR Development Partnership. Technical Overview V2.2.1, February 2008.
http://www.autosar.org.

[9] Ji Eun Kim, Rahul Kapoor, Martin Herrmann, Jochen Haerdtlein, Franz Grzeschniok, and Peter Lutz. Software Behavior Description of Real-Time Embedded Systems in Component Based Software Development. In *ISORC '08: Proceedings of the 2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing*, pages 307–311, Washington, DC, USA, 2008. IEEE Computer Society.

[10] Ji Eun Kim, Oliver Rogalla, Simon Kramer, and Arne Haman. Extracting, Specifying and Predicting Software System Properties in Component Based Real-Time Embedded Software Development. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, 2009.

[11] Arcticus Systems. Rubus Software Components.
http://www.arcticus-systems.com.

[12] Rob van Ommering, Frank van der Linden, Jeff Kramer, and Jeff Magee. The Koala Component Model for Consumer Electronics Software. *Computer*, 33(3):78–85, 2000.

[13] Oscar Nierstrasz, Gabriela Arévalo, Stéphane Ducasse, Roel Wuyts, Andrew P. Black, Peter O. Müller, Christian Zeidler, Thomas Genssler, and Reinier van den Born. A Component Model for Field Devices. In *Proc. of the 1st Int. IFIP/ACM Working Conference on Component Deployment*, pages 200–209. Springer, 2002.

[14] Mikael Åkerholm, Jan Carlson, Johan Fredriksson, Hans Hansson, John Håkanssonm, Anders Möller, Paul Pettersson, and Massimo Tivoli. The SAVE Approach to Component-Based Development of Vehicular Systems. *Journal of Systems and Software*, 80(5):655–667, May 2007.

[15] H. Maaskant. A Robust Component Model for Consumer Electronic Products. In *Dynamic and Robust Streaming in and between Connected Consumer-Electronic Devices*, volume 3 of *Philips Research*, pages 167–192. Springer, 2005.

[16] Scott Hissam, Gabriel Moreno, Judith Stafford, and Kurt Wallnau. Packaging Predictable Assembly with Prediction-Enabled Component Technology. Technical report, 2001.

[17] Ivica Crnkovic, Séverine Sentilles, Aneta Vulgarakis, and Michel Chaudron. A Classification Framework for Software Component Models. *IEEE Transaction of Software Engineering*, 37(5):593–615, October 2011.

[18] Séverine Sentilles, Petr Stepan, Jan Carlson, and Ivica Crnkovic. Integration of Extra-Functional Properties in Component Models. In Iman Poernomo Christine Hofmeister, Grace A. Lewis, editor, *12th International Symposium on Component Based Software Engineering (CBSE 2009), LNCS 5582*. Springer Berlin, LNCS 5582, June 2009.

[19] Aneta Vulgarakis, Séverine Sentilles, Jan Carlson, and Cristina Seceleanu. Integrating Behavioral Descriptions into a Component Model for Embedded Systems. In *36th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 113–118. IEEE, September 2010.

[20] Thomas Lévêque and Séverine Sentilles. Refining Extra-Functional Property Values in Hierarchical Component Models. In *The 14th International ACM SIGSOFT Symposium on Component Based Software Engineering (CBSE-2011)*. ACM SIGSOFT, June 2011.

[21] Tomáš Bureš, Jan Carlson, Séverine Sentilles, and Aneta Vulgarakis. A Component Model Family for Vehicular Embedded Systems. In *The Third International Conference on Software Engineering Advances*. IEEE, October 2008.

[22] Séverine Sentilles, Aneta Vulgarakis, Tomáš Bureš, Jan Carlson, and Ivica Crnković. A Component Model for Control-Intensive Distributed Embedded Systems. In Michel R.V. Chaudron and Clemens Szyperski, editors, *Proceedings of the 11th International Symposium on Component Based Software Engineering (CBSE2008)*, pages 310–317. Springer Berlin, October 2008.

[23] Séverine Sentilles, John Håkansson, Paul Pettersson, and Ivica Crnkovic. Save-IDE – An Integrated Development Environment for Building Predictable Component-Based Embedded Systems. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008)*, September 2008.

[24] Séverine Sentilles, Anders Pettersson, Dag Nyström, Thomas Nolte, Paul Pettersson, and Ivica Crnkovic. Save-IDE — A Tool for Design,

Analysis and Implementation of Component-Based Embedded Systems. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, May 2009.

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

[26] Mary Shaw. Writing Good Software Engineering Research Papers. In *Proceedings of the 25th International Conference on Software Engineering*, pages 726–736, 2003.

[27] Michel Chaudron and Ivica Crnkovic. *Software Engineering: Principles and Practice, 3rd Edition*, chapter 18 in H. van Vliet, Component-Based Software Engineering. Wiley, 2008.

[28] Ivica Crnkovic, Magnus Larsson, and Otto Preiss. Concerning Predictability in Dependable Component-Based Systems: Classification of Quality Attributes. In R. de Lemos et al. (Eds.):, editor, *WADS'04*, page pp. 257 278. Springer, LNCS 3549, 2005.

[29] Ivica Crnkovic, Michel Chaudron, and Stig Larsson. Component-based Development Process and Component Lifecycle. *Journal of Computing and Information Technology*, 13(4):321–327, November 2005.

[30] John Cheesman and John Daniels. *UML components: a simple process for specifying component-based software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[31] Colin Atkinson, Joachim Bayer, Christian Bunse, Erik Kamsties, Oliver Laitenberger, Roland Laqua, Dirk Muthig, Barbara Paech, Jürgen Wüst, and Jörg Zettel. *Component-Based Product Line Engineering with UML*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[32] Scott Hissam, James Ivers, Daniel Plakosh, and Kurt C. Wallnau. Pin Component Technology (V1.0) and Its C Interface. Technical Note: CMU/SEI-2005-TN-001, April 2005.

[33] EJB 3.0 Expert Group. JSR 220: Enterprise JavaBeansTM,Version 3.0 EJB Core Contracts and Requirements Version 3.0, Final Release, May 2006.

[34] OMG CORBA Component Model v4.0. Available at `www.omg.org/docs/formal/06-04-01.pdf`.

[35] Dale Rogerson. *Inside COM*. Microsoft Press, 1997.

[36] Oxford Advanced Learners Dictionary.

[37] Sun Microsystems. JavaBeans Specification, 1997.

[38] OSGi Alliance. OSGi Service Plaform Core Specification, V4.1, 2007.

[39] The Object Management Group. UML Superstructure Specification v2.1, April 2009. `http://www.omg.org/docs/ptc/06-04-02.pdf`.

[40] IEC. Application and Implementation of IEC 61131-3. IEC, 1995.

[41] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making Components Contract Aware. *Computer*, 32(7):38–45, 1999.

[42] George T. Heineman and William T. Councill. *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley Longman Publishing Co., 2001.

[43] Eric Bruneton, Thierry Coupaye, and Jean-Bernard Stefani. The Fractal Component Model Specification. *The ObjectWeb Consortium, Tech. Rep., February*, 2004.

[44] Ananda Basu, Marius Bozga, and Joseph Sifakis. Modeling Heterogeneous Real-time Components in BIP. In *Proc. of the 4th IEEE International Conference on Software Engineering and Formal Methods*, pages 3–12. IEEE, 2006.

[45] Wolfgang Emmerich, Mikio Aoyama, and Joe Sventek. The Impact of Research on the Development of Middleware Technology. *ACM Trans. Softw. Eng. Methodol.*, 17(4):19:1–19:48, August 2008.

[46] Mary Shaw. Truth vs Knowledge: The Difference Between What a Component Does and What We Know It Does. *International Workshop on Software Specification and Design*, page 181, 1996.

[47] PECT homepage. `www.sei.cmu.edu/pacc/pect_init.html`.

[48] S Becker, H Koziolek, and R Reussner. Model-Based Performance Prediction with the Palladio Component Model. *the 6th international workshop on Software and performance*, 2007.

[49] Nenad Medvidovic and Richard N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Trans. Softw. Eng.*, 26(1):70–93, January 2000.

[50] Kung-Kiu Lau and Zheng Wang. Software Component Models. *IEEE Transactions on Software Engineering*, 33(10):709–724, 2007.

[51] Stephen J. Mellor and Marc Balcer. *Executable UML: A Foundation for Model-Driven Architectures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. Foreword By-Jacoboson, Ivar.

[52] Hongyu Pei Breivold and Magnus Larsson. Component-Based and Service-Oriented Software Engineering: Key Concepts and Principles. pages 13–20, Aug. 2007.

[53] Xu Ke, Krzysztof Sierszecki, and Christo Angelov. COMDES-II: A Component-Based Framework for Generative Development of Distributed Real-Time Control Systems. In *Proc. of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 199–208. IEEE, 2007.

[54] Rémi Bastide and Eric Barboni. Component-Based Behavioural Modelling with High-Level Petri Nets . In *MOCA '04 - Third Workshop on Modelling of Objects, Components and Agents , Aahrus, Denmark , 11/10/04-13/10/04*, pages 37–46. DAIMI, octobre 2004.

[55] IEC. IEC 61499 Function Blocks for Embedded and Distributed Control Systems Design. IEC, 2005.

[56] Michael Clarke, Gordon S. Blair, Geoff Coulson, and Nikos Parlavantzas. An efficient Component Model for the Construction of Adaptive Middleware. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, Middleware '01, pages 160–178, London, UK, UK, 2001. Springer-Verlag.

[57] M Winter, C Zeidler, and C Stich. The PECOS software process. *Workshop on Components-based Software Development Processes, ICSR*, 2002.

[58] Johan Muskens, Michel R.V. Chaudron, and Johan J. Lukkien. *Component-Based Software Development for Embedded Systems*, chapter A Component Framework for Consumer Electronics Middleware, pages 164–184. Springer Verlag, 2005.

[59] Tomáš Bureš, Petr Hnětynkal, and František Plášil. SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model. In *Proceedings of the Fourth International Conference on Software Engineering Research, Management and Applications*, SERA '06, pages 40–48, Washington, DC, USA, 2006. IEEE Computer Society.

[60] Ivica Crnkovic, Aneta Vulgarakis, Mario Zagar, Ana Petricic, Juraj Feljan, Luka Lednicki, and Josip Maras. Classification and Survey of Component Models. In *DICES workshop @ SoftCOM 2010*, September 2010.

[61] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Professional, December 1997.

[62] Nenad Medvidovic, Eric M. Dashofy, and Richard N. Taylor. Moving Architectural Description from under the Technology Lamppost. *Inf. Softw. Technol.*, 49(1):12–31, 2007.

[63] Gerald Kotonya, Ian Sommerville, and Steve Hall. Towards A Classification Model for Component-Based Software Engineering Research. In *EUROMICRO '03: Proceedings of the 29th Conference on EUROMICRO*, page 43, Washington, DC, USA, 2003. IEEE Computer Society.

[64] Andreas Ermedahl. *A Modular Tool Architecture for Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University: Acta Universitatis Upsaliensis, June 2003.

[65] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The Worst-Case Execution-Time problem — Overview of Methods and Survey of Tools. *ACM Trans. Embed. Comput. Syst.*, 7:36:1–36:53, May 2008.

[66] ISO/IEC. Information Technology - Software product quality - Part 1: Quality model. Report: ISO/IEC FDIS 9126-1:2000, 2000.

[67] Manuel F. Bertoa and Antonio Vallecillo. Quality attributes for COTS components. In *6th International Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE'2002*, 2002.

[68] Antonio Cicchetti, Federico Ciccozzi, Thomas Lévêque, and Séverine Sentilles. Evolution Management of Extra-Functional Properties in Component-Based Embedded Systems . In *The 14th International ACM SIGSOFT Symposium on Component Based Software Engineering (CBSE-2011)*. ACM SIGSOFT, June 2011.

[69] Reidar Conradi and Bernhard Westfechtel. Version Models for Software Configuration Management. *ACM Comput. Surv.*, 30:232–282, June 1998.

[70] Object Management Group. OMG Systems Modeling Language, V1.0, 2007.

[71] Object Management Group. UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems. OMG Document Number: formal/2011-06-02, June 2011.

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

[73] Aneta Vulgarakis, Jagadish Suryadevara, Jan Carlson, Cristina Seceleanu, and Paul Pettersson. Formal Semantics of the ProCom Real-Time Component Model. In *Proceedings of the 35th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, August 2009.

[74] Cristina Seceleanu, Aneta Vulgarakis, and Paul Pettersson. REMES: A Resource Model for Embedded Systems. Technical Report ISSN 1404-3041 ISRN MDH-MRTC-232/2008-1-SE, Mälardalen University, October 2008.

[75] John Håkansson and Paul Pettersson. Partial Order Reduction for Verification of Real-Time Components. In Jean-Franois Raskin and P.S.

Thiagarajan, editors, *Proceedings of the 5th International Conference on Formal Modelling and Analysis of Timed Systems, Lecture Notes in Computer Science 4763*, pages 211–226. Springer Verlag, October 2007.

[76] Mikael Åkerholm, Jan Carlson, John Håkansson, Hans Hansson, Mikael Nolin, Thomas Nolte, and Paul Pettersson. The SaveCCM language reference manual. Technical Report MDH-MRTC-207/2007-1-SE, Mälardalen University, January 2007.

[77] Mikael Åkerholm, Jan Carlson, John Håkansson, Hans Hansson, Mikael Nolin, Thomas Nolte, and Paul Pettersson. The SaveCCM Language Reference Manual. Technical Report ISSN 1404-3041 ISRN MDH-MRTC-207/2007-1-SE, Mälardalen University, January 2007.

[78] UPPAAL. www.uppaal.com, accessed March 2007.

[79] John Håkansson, Jan Carlson, Aurelien Monot, and Paul Pettersson. Component-Based Design and Analysis of Embedded Systems with UPPAAL PORT. In Moonzoo Kim Insup Lee Mahesh Viswanathan Sungdeok Cha, Jin-Young Choi, editor, *6th International Symposium on Automated Technology for Verification and Analysis*, pages 252–257. Springer-Verlag, October 2008.

[80] CC Systems AB. CCSimTech. http://www.cc-systems.com/.

[81] Ana Petricic, Luka Lednicki, and Ivica Crnkovic. An Empirical Comparison of SaveUML and SaveCCM Technologies. Technical Report, Mälardalen University, March 2009.

[82] Davor Slutej, John Håkansson, Jagadish Suryadevara, Cristina Seceleanu, and Paul Pettersson. Analyzing a Pattern-Based Model of a Real-Time Turntable System. In *6th International Workshop on Formal Engineering approaches to Software Components and Architectures(FESCA), ETAPS 2009, York, UK*. Electronic Notes in Theoretical Computer Science (ENTCS), Elsevier, March 2009.

[83] Thomas Lévêque, Etienne Borde, Amine Marref, and Jan Carlson. Hierarchical Composition of Parametric WCET in a Component Based Approach. In *14th IEEE International Symposium on Object/Component/Service-oriented Real-time Distributed Computing (ISORC'11)*. IEEE, March 2011.

[84] Dinko Ivanov, Marin Orlic, Cristina Seceleanu, and Aneta Vulgarakis. REMES Tool-chain - A Set of Integrated Tools for Behavioral Modeling and Analysis of Embedded Systems. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE 2010)*, September 2010.

[85] Malcolm Wallace. Modular Architectural Representation and Analysis of Fault Propagation and Transformation. *Electr. Notes Theor. Comput. Sci.*, 141(3):53–71, 2005.

[86] Thomas Lévêque, Jan Carlson, Séverine Sentilles, and Etienne Borde. Flexible Semantic-Preserving Flattening of Hierarchical Component Models. In *37th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE Computer Society, August 2011.

[87] V. Bos and J. J. T. Kleijn. Automatic verification of a manufacturing system. *Robotics and Computer-Integrated Manufacturing*, 17(3):185–198, 2001.

[88] Elena M. Bortnik, Nikola Trčka, Anton Wijs, Bas Luttik, J. M. van de Mortel-Fronczak, Jos C. M. Baeten, Wan Fokkink, and J. E. Rooda. Analyzing a χ model of a turntable system using Spin, CADP and Uppaal. *Journal of Logic and Algebraic Programming*, 65(2):51–104, 2005.

[89] Garmin. GPS 18 Technical Specifications (190-00307-00), Rev. D. Technical report, June 2005.

[90] Nissan. Distance Control Assist System. `www.nissan-global.com/EN/TECHNOLOGY/OVERVIEW/dcas.html`, last accessed: 02/05/2012.

[91] Jan Carlson. Timing Analysis of Component-based Embedded Systems. In *15th International ACM SIGSOFT Symposium on Component Based Software Engineering*. ACM, June 2012.

[92] Tidorum Ltd. Bound-T time and stack analyser. `www.bound-t.com/`, last accessed: 15/05/2012.

[93] Sherif Yacoub, Hany Ammar, and Ali Mili. A Model for Classifying Component Interfaces. In *Second International Workshop on Component-Based Software Engineering, in conjunction with the 21 st*

International Conference on Software Engineering (ICSE99, pages 17–18, 1999.

[94] Sherif Yacoub, Hany Ammar, and Ali Mili. Characterizing a Software Component. In *In Proceedings of the 2nd Workshop on Component-Based Software Engineering, in conjunction with ICSE99*, 1999.

[95] Klement J. Fellner and Klaus Turowski. Classification Framework for Business Components. In *HICSS '00: Proceedings of the 33rd Hawaii International Conference on System Sciences-Volume 8*, page 8047, Washington, DC, USA, 2000. IEEE Computer Society.

[96] Barbara Kitchenham, Shari Lawrence Pfleeger, Lesley Pickard, Peter Jones, David C. Hoaglin, Khaled El Emam, and Jarrett Rosenberg. Preliminary Guidelines for Empirical Research in Software Engineering. *IEEE Trans. Software Eng.*, 28(8):721–734, 2002.

[97] Heinz Schmidt. Trustworthy components–compositionality and prediction. *Journal of Systems and Software*, 65(3):215 – 225, 2003. Component-Based Software Engineering.

[98] Steffen Zschaler. Formal Specification of Non-functional Properties of Component-Based Software Systems: A Semantic Framework and Some Applications Thereof. *Software and Systems Modelling (SoSyM)*, 9:161–201, April 2009.

[99] Jim A. Mccall, Paul K. Richards, and Gene F. Walters. Factors in Software Quality. Volume I. Concepts and Definitions of Software Quality. Technical Report ADA049014, GENERAL ELECTRIC CO SUNNYVALE CALIF, November 1977.

[100] B. W. Boehm, J. R. Brown, and M. Lipow. Quantitative evaluation of software quality. In *Proceedings of the 2nd international conference on Software engineering*, ICSE '76, pages 592–605, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[101] Mario Barbacci, Mark H. Klein, Thomas A. Longstaff, and Charles B. Weinstock. Quality Attributes. Technical report CMU/SEI-95-TR-021, Software Engineering Institute, 1995.

[102] ISO/IEC. Software engineering - Software product Quality Requirements and Evaluation (SQuaRE) Quality model. Report: ISO/IEC 25010:2011, 2011.

[103] Re Alvaro, Eduardo Santana De Almeida, Silvio Romero, and Lemos Meira. Quality Attributes for a Component Quality Model. In *In the 10th International Workshop on Component-Oriented Programming (WCOP) in Conjunction with the 19th European Conference on Object Oriented Programming (ECOOP*, 2005.

[104] Martin Glinz. On Non-Functional Requirements. In *Requirements Engineering Conference, 2007. RE '07. 15th IEEE International*, pages 21 –26, oct. 2007.

[105] Information Technology – Quality of Service: Framework. ISO/IEC 13236:1998, ITU-T X.641, 1998.

[106] Pere Botella, Xavier Franch, and Xavier Burgus. Using Non-Functional Requirements in Component-Based Software Construction, 1996.

[107] Soo Dong Kim, Jin Sun Her, and Soo Ho Chang. A theoretical foundation of variability in component-based development. *Inf. Softw. Technol.*, 47(10):663–673, July 2005.

[108] Svend Frølund and Jari Koistinen. Quality of services specification in distributed object systems design. In *Proceedings of the 4th conference on USENIX Conference on Object-Oriented Technologies and Systems - Volume 4*, COOTS'98, pages 1–1, Berkeley, CA, USA, 1998. USENIX Association.

[109] Svend Frølund and Jari Koistinen. QML: A Language for Quality of Service Specification. Technical report, HPL-98-10, February 1998.

[110] Jean-Marc Jézéquel, Olivier Defour, and Noël Plouzeau. An MDA Approach to Tame Component Based Software Development. In *FMCO*, pages 260–275, 2003.

[111] Jan Øyvind Aagedal. *Quality of Service Support in Development of Distributed Systems*. PhD thesis, Faculty of Mathematics and Natural Sciences, University of Oslo, 2001.

[112] Olivier Defour, Jean-Marc Jézéquel, and Noël Plouzeau. Extra-Functional Contract Support in Components. In *CBSE*, pages 217–232, 2004.

[113] Simone Röttger and Steffen Zschaler. CQML+: Enhancements to CQML. In Jean-Michel Bruel, editor, *Proc. 1st Int'l Workshop on Quality of Service in Component-Based Software Engineering, Toulouse, France*, pages 43–56. Cépaduès-Éditions, June 2003.

[114] The Object Management Group. UML Profile for Schedulability, Performance and Time, January 2005.

[115] Object Management Group. UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms Specification, version 1.1. OMG Document Number: formal/2008-04-05, April 2008.

[116] Steffen Becker, Heiko Koziolek, and Ralf Reussner. The Palladio Component Model for Model-driven Performance Prediction. *Journal of Systems and Software*, 82:3–22, 2009.

[117] Scott Hissam, Gabriel Moreno, Judith Stafford, and Kurt Wallnau. Enabling predictable assembly. *J. Syst. Softw.*, 65(3):185–198, March 2003.

[118] Kurt C. Wallnau. Volume III: A Technology for Predictable Assembly from Certifiable Components. Technical report, April 2003.

[119] Sagar Chaki, James Ivers, Natasha Sharygina, and Kurt Wallnau. The ComFoRT reasoning framework. In *Proceedings of the 17th international conference on Computer Aided Verification*, CAV'05, pages 164–169, Berlin, Heidelberg, 2005. Springer-Verlag.

[120] Scott Hissam and Sagar Chaki. Certifying the Absence of Buffer Overflows. Technical report, September 2006.

[121] Gabriel A. Moreno and Jeffrey Hansen. Overview of the Lambda-* Performance Reasoning Frameworks. Technical report, February 2009.

[122] Kurt C. Wallnau and Judith A. Stafford. Ensembles: Abstractions for a New Class of Design Problem. In *EUROMICRO*, pages 48–55, 2001.

[123] Steffen Zschaler. *A Semantic Framework for Non-functional Specifications of Component-Based Systems*. Dissertation, Technische Universität Dresden, Dresden, Germany, April 2007.

[124] Steffen Zschaler. Formal Specification of Non-Functional Properties of Component-Based Software. In *In: Proc. Workshop on Models for Non-functional Aspects of Component-Based Systems*, 2004.

[125] Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[126] Kamil Jezek, Premek Brada, and Petr Stepan. Towards Context Independent Extra-functional Properties Descriptor for Components. *Electronic Notes in Theoretical Computer Science*, 264(1):55–71, 2010.

[127] Kamil Ježek and Přemek Brada. Correct Matching of Components with Extra-functional Properties – A Framework Applicable to a Variety of Component Models. In *Evaluation of Novel Approaches to Software Engineering (ENASE 2011)*, 2011.

[128] H. Fennel et al. Achievements and Exploitation of the AUTOSAR Development Partnership. Presented at Convergence 2006, Detroit, MI, USA, October 2006.
`http://www.autosar.org`.

[129] M. Jersak et.al. Timing Model and Methodology for AUTOSAR. *Elektronik automotive, Special issue AUTOSAR*, 2007.

[130] Philippe Cuenot, DeJiu Chen, Sebastien Gerard, Henrik Lonn, Mark-Oliver Reiser, David Servat, Carl-Johan Sjostedt, Ramin Tavakoli Kolagari, Martin Torngren, and Matthias Weber. Managing Complexity of Automotive Electronics Using the EAST-ADL. In *ICECCS '07: Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems*, pages 353–358, Washington, DC, USA, 2007. IEEE Computer Society.

[131] Bernhard F. Weichel and Martin Herrmann. A Backbone in Automotive Software Development Based on Xml and Asam/Msr. SAE World Congress, 2004.

[132] Michael Winter, Thomas Genler, Alexander Christoph, Oscar Nierstrasz, Stephane Ducasse, Roel Wuyts, Gabriela Arevalo, Peter Mller, Chris Stich, and Bastiaan Schönhage. Components for Embedded Software - The PECOS Approach. In *In Proc. International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES 02*. ACM Press, 2002.

[133] Peter H. Feiler, Bruce Lewis, and Steve Vestal. The SAE architecture analysis & design language (AADL) standard: A Basis for Model-Based Architecture-Driven Embedded Systems Engineering. *Proceeding of the RTAS 2003 Workshop*, 2003.

[134] Peter H. Feiler, David P. Gluch, and John J. Hudak. The Architecture Analysis & Design Language (AADL): An Introduction. Technical report, Software Engineering Institute, 2006.

[135] Mikael Åkerholm. *Reusability of Software Components in the Vehicular Domain*. PhD thesis, Mälardalen University Press, May 2008.

[136] Luka Lednicki, Juraj Feljan, Jan Carlson, and Mario Zagar. Adding Support for Hardware Devices to Component Models for Embedded Systems. In *ICSEA 2011, The Sixth International Conference on Software Engineering Advances*, pages 149–154. IARIA, October 2011.

[137] Jan Carlson, Juraj Feljan, Jukka Mäki-Turja, and Mikael Sjödin. Deployment Modelling and Synthesis in a Component Model for Distributed Embedded Systems. In *36th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, September 2010.