Doctoral Thesis

Automated instantiation of software frameworks for control systems

Author(s): Čechtický, Václav

Publication Date: 2004

Permanent Link: https://doi.org/10.3929/ethz-a-004923956

Rights / License: In Copyright - Non-Commercial Use Permitted
Automated Instantiation of Software Frameworks for Control Systems

A dissertation submitted to the
Swiss Federal Institute of Technology Zürich

for the degree of
Doctor of Technical Sciences

presented by
Václav Čechtický
Dipl. El.-Ing. CTU
Czech Technical University, Prague
born July 30th, 1974
citizen of Czech Republic

accepted on the recommendation of
Prof. Dr. Walter Schaufelberger, examiner
Prof. Dr. Wolfgang Pree, co-examiner
Dr. Alessandro Pasetti, co-examiner

2004
This thesis is the result of research work I carried out at the Institute für Automatik of the Swiss Federal Institute of Technology (ETH) in Zürich. First and foremost, I would like to express my sincere gratitude to my advisor Prof. Walter Schaufelberger for giving me the opportunity and scientific freedom to work on challenging tasks. I thank him for the interesting discussions and encouragement that have helped to get the work to a successful end.

I would like to thank my co-examiner Prof. Wolfgang Pree for his interest and remarks.

I would like express a special thank to Alessandro Pasetti for his interest in my work, his guidance and support, but also for his personal advice and help. I greatly appreciate valuable work of all the students who contributed to this thesis, particularly: Alex Blum, Florian Hanke, and Charly Wilhelm. My special thank goes to my colleagues both in Zürich and my former department in Prague who have contributed or supported me: Franta Kraus, Marco Sanvido, Konrad Stadler, Ondřej Rohlik, Michal Kvasnica, Iwan Birrer & Karin Brütsch, Domenico Mignone, Francesco Borrelli, Sašo Jezernik, Marc Lawrence, Anders Hagström, Frank Christophersen, Valentina Sartori, Eleonora Zanderigo, Petr Horáček, and Vladimír Havlena. I would like to thank the administrative staff of the Institute für Automatik including Esther Hagenow, Martine D’Emma, Alice Vyskocil, Danielle Couson and Gabriela Hess.

Last but not least I would like to thank to my family for their support throughout many years of education.

Thank you!
Abstract

This thesis introduces generic concepts for design, implementation and instantiation of software frameworks. The concepts were successfully applied in the control system domain.

Control systems traditionally rely on conservative solutions, and are therefore a step back behind the latest and progressive software engineering developments. Software frameworks can be considered as a representation of successful progressive software engineering techniques, which are commonly used today for the rapid development of reliable software applications for a predefined domain.

This thesis contributes to the problem of software framework design, implementation and instantiation in several respects. First, it provides a design and implementation of a software framework covering the control system domain. Second, it evaluates Java object-oriented programming language for purposes of real-time programming. A real-time subset of Java was used for the framework implementation. Third, it proposes a concept of automated framework instantiation process, which can provide the user fully automated code generation. Fourth, it proposes a technique that improves traditional feature modelling approaches and provides instantiation facilities. Finally, it demonstrates the usage of the new feature-based modelling and automated instantiation process on concrete examples. Note that the distinctive characteristic of the new instantiation approaches is their simplicity and reliance only on mainstream technologies and tools, which might imply in acceptance of the instantiation concepts among software framework designers and users.

Keywords: control systems, software framework, instantiation process, code generation, Java

Regelungssysteme basieren traditionell auf konservativen Lösungen und hinken daher den neuesten Entwicklungen der Softwaretechnik hinterher. Software Frameworks können als eine erfolgreiche Repräsentation von fortschrittlichen Softwareentwicklungstechniken gesehen werden. Diese werden heute für die beschleunigte Entwicklung von zuverlässigen Applikationen in einer definierten Anwendungsumgebung eingesetzt.

ware Frameworks erhöhen.

**Keywords:** Regelungssysteme, Software Framework, Instantiierungsprozess, Code Generierung, Java
3.3.1 Approaches to Real-Time Java .............................................. 50

3.4 Implementation Aspects of the RTJ Framework .............................. 54

3.4.1 Framework Components as JavaBeans ........................................ 55

3.4.2 Real-Time Issues ........................................................................ 55

3.4.3 Scheduling and Synchronization .................................................. 56

3.4.4 Framework Memory Model .......................................................... 57

3.4.5 Integration of Non-Standard Components ..................................... 58

3.4.6 Testing ......................................................................................... 59

3.4.7 Documentation Approach ............................................................ 62

3.5 Conclusions ................................................................................. 62

4 Introduction to Instantiation Problem .................................................. 65

4.1 Introduction .................................................................................... 66

4.2 Motivation ....................................................................................... 66

4.3 Instantiation Process ........................................................................ 66

4.4 Instantiation Approaches ................................................................. 67

4.4.1 Naive Instantiation Approach ...................................................... 67

4.4.2 Factory Instantiation Approach ................................................... 67

4.4.3 Automated Instantiation Approach .............................................. 71

4.4.4 Feature-Based Instantiation Approach ....................................... 72

4.5 Conclusions ................................................................................... 72

5 Automated Instantiation ..................................................................... 73

5.1 Introduction .................................................................................... 74

5.2 Automated Instantiation Environment ............................................ 74
7.1 Introduction .......................................................... 111
7.2 Motivation ............................................................ 111
7.3 Overview of Test Cases .............................................. 111
7.4 Controlled Plants .................................................... 112
  7.4.1 The Swing Mass Model ......................................... 112
  7.4.2 The ER1 Robot .................................................. 112
7.5 Platforms ............................................................ 114
  7.5.1 Jbed & PowerPC ................................................. 114
  7.5.2 PERC & VxWorks & PC ........................................ 115
  7.5.3 Standard Java & PC ............................................. 116
7.6 Instantiated Control Applications .................................. 116
  7.6.1 The Control Application for the Swing Mass Model Deployed on the Jbed Platform .............................................. 116
  7.6.2 The Control Application for the Swing Mass Model Deployed on the PERC Platform ............................................. 121
  7.6.3 The Control Application for the ER1 Robot Deployed on the Java Platform ..................................................... 126
7.7 Factory Instantiation Approach ...................................... 129
  7.7.1 The Control Application for the Swing Mass Model Deployed on the Jbed Platform .............................................. 129
  7.7.2 The Control Application for the Swing Mass Model Deployed on the PERC Platform ............................................. 131
  7.7.3 The Control Application for the ER1 Robot Deployed on the Java Platform ..................................................... 132
7.8 Automated Instantiation Approach ................................... 133
  7.8.1 The Control Application for the Swing Mass Model Deployed on the Jbed Platform .............................................. 133
7.8.2 The Control Application for the ER1 Robot Deployed on the Java Platform ........................................ 135

7.9 Feature-Based Instantiation Approach ........................................ 137

7.9.1 The Control Application for the Swing Mass Model Deployed on the Jbed Platform ........................................ 138

7.10 Conclusions ........................................ 139

8 Conclusion ........................................ 141

8.1 Summary ........................................ 142

8.2 Achieved Goals ........................................ 142

8.3 Main Contributions ........................................ 143

8.4 Future Work ........................................ 144

A Description of Control Applications ........................................ 145

A.1 The Control Application for the Swing Mass Model Deployed on the Jbed Platform ........................................ 145

A.1.1 Operational Modes ........................................ 145

A.1.2 Data Pools ........................................ 146

A.1.3 Failure Detection Policy ........................................ 146

A.1.4 Failure Recovery Policy ........................................ 147

A.1.5 Telecommanding Policy ........................................ 147

A.1.6 Telemetry Policy ........................................ 148

A.1.7 Manoeuvres ........................................ 149

A.1.8 Controllers ........................................ 149

A.1.9 Sensors and Actuators ........................................ 149

A.1.10 External Interfaces ........................................ 149
A.2 The Control Application for the Swing Mass Model Deployed on the PERC Platform .............................................. 149

A.2.1 Operational Modes .............................................. 149
A.2.2 Data Pools ...................................................... 150
A.2.3 Failure Detection Policy ....................................... 151
A.2.4 Failure Recovery Policy ....................................... 151
A.2.5 Telecommanding Policy ....................................... 152
A.2.6 Telemetry Policy ............................................... 152
A.2.7 Manoeuvres .................................................... 153
A.2.8 Controllers ..................................................... 154
A.2.9 Sensors and Actuators ......................................... 154
A.2.10 External Interfaces ........................................... 154

A.3 The Control Application for the ER1 Robot Deployed on the Java Platform ......................................................... 155

A.3.1 Operational Modes .............................................. 155
A.3.2 Data Pools ...................................................... 155
A.3.3 Failure Detection Policy ....................................... 155
A.3.4 Failure Recovery Policy ....................................... 156
A.3.5 Telecommanding Policy ....................................... 156
A.3.6 Telemetry Policy ............................................... 156
A.3.7 Manoeuvre ...................................................... 157
A.3.8 Controllers ..................................................... 157
A.3.9 Sensors and Actuators ......................................... 157
A.3.10 External Interfaces ........................................... 158

B Application Factories and Components .......................... 159
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.1</td>
<td>The Control Application for the Swing Mass Model Deployed on the Jbed Platform</td>
<td>159</td>
</tr>
<tr>
<td>B.1.1</td>
<td>Application Specific Factories</td>
<td>159</td>
</tr>
<tr>
<td>B.1.2</td>
<td>Application Specific Components</td>
<td>160</td>
</tr>
<tr>
<td>B.2</td>
<td>The Control Application for the Swing Mass Model Deployed on the PERC Platform</td>
<td>162</td>
</tr>
<tr>
<td>B.2.1</td>
<td>Application Specific Factories</td>
<td>162</td>
</tr>
<tr>
<td>B.2.2</td>
<td>Application Specific Components</td>
<td>163</td>
</tr>
<tr>
<td>B.3</td>
<td>The Control Application for the ER1 Robot Deployed on the Java Platform</td>
<td>165</td>
</tr>
<tr>
<td>B.3.1</td>
<td>Application Specific Factories</td>
<td>165</td>
</tr>
<tr>
<td>B.3.2</td>
<td>Application Specific Components</td>
<td>166</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>1.1</td>
<td>Functional aspects of control systems.</td>
<td>16</td>
</tr>
<tr>
<td>2.1</td>
<td>The adaptation mechanism based on inheritance.</td>
<td>23</td>
</tr>
<tr>
<td>2.2</td>
<td>The adaptation mechanism based on object composition.</td>
<td>24</td>
</tr>
<tr>
<td>2.3</td>
<td>The software development process based on domain and application engineering [19, 27].</td>
<td>26</td>
</tr>
<tr>
<td>2.4</td>
<td>The framework development and the application development processes.</td>
<td>27</td>
</tr>
<tr>
<td>2.5</td>
<td>The connection of a real unit, fictitious unit and final user.</td>
<td>36</td>
</tr>
<tr>
<td>2.6</td>
<td>The feature model of the Unit Management Architecture.</td>
<td>37</td>
</tr>
<tr>
<td>2.7</td>
<td>The customized Command design pattern.</td>
<td>39</td>
</tr>
<tr>
<td>2.8</td>
<td>The customized Proxy design pattern.</td>
<td>39</td>
</tr>
<tr>
<td>2.9</td>
<td>The customized Strategy design pattern.</td>
<td>40</td>
</tr>
<tr>
<td>2.10</td>
<td>The Unit abstract model.</td>
<td>42</td>
</tr>
<tr>
<td>2.11</td>
<td>The gain block from the Simulink library.</td>
<td>44</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>2.12</td>
<td>The customized Unification design pattern</td>
<td>44</td>
</tr>
<tr>
<td>3.1</td>
<td>Roads to real-time Java</td>
<td>52</td>
</tr>
<tr>
<td>3.2</td>
<td>Jbed execution model</td>
<td>54</td>
</tr>
<tr>
<td>3.3</td>
<td>PERC execution model</td>
<td>55</td>
</tr>
<tr>
<td>3.4</td>
<td>The component testing – a customized Strategy design pattern</td>
<td>59</td>
</tr>
<tr>
<td>4.1</td>
<td>The instantiator concept</td>
<td>68</td>
</tr>
<tr>
<td>4.2</td>
<td>The instantiation factories</td>
<td>70</td>
</tr>
<tr>
<td>4.3</td>
<td>The sequence diagram of the instantiation process</td>
<td>70</td>
</tr>
<tr>
<td>5.1</td>
<td>The automated instantiation approach</td>
<td>75</td>
</tr>
<tr>
<td>5.2</td>
<td>The component encoding process</td>
<td>76</td>
</tr>
<tr>
<td>5.3</td>
<td>The transformation of visual proxy description to JavaBeans component model</td>
<td>77</td>
</tr>
<tr>
<td>5.4</td>
<td>The customized component composition environment</td>
<td>78</td>
</tr>
<tr>
<td>5.5</td>
<td>The instantiation code generation process</td>
<td>80</td>
</tr>
<tr>
<td>5.6</td>
<td>An example of automatically generated instantiation code</td>
<td>81</td>
</tr>
<tr>
<td>5.7</td>
<td>The simulation concept</td>
<td>82</td>
</tr>
<tr>
<td>5.8</td>
<td>An evaluation of the repository check simulation</td>
<td>83</td>
</tr>
<tr>
<td>5.9</td>
<td>Selection of components and their methods and properties that should be simu-</td>
<td>85</td>
</tr>
<tr>
<td></td>
<td>lated</td>
<td></td>
</tr>
<tr>
<td>5.10</td>
<td>The simulator</td>
<td>86</td>
</tr>
<tr>
<td>5.11</td>
<td>The generic instantiation approach</td>
<td>87</td>
</tr>
<tr>
<td>5.12</td>
<td>The feature diagram of the framework description model</td>
<td>89</td>
</tr>
<tr>
<td>5.13</td>
<td>The framework description model [49]</td>
<td>90</td>
</tr>
<tr>
<td>5.14</td>
<td>The graphical instantiation environment architecture</td>
<td>90</td>
</tr>
</tbody>
</table>
List of Figures

5.15 The GIE user interface. ................................................................. 91

6.1 A feature model example. ............................................................ 96

6.2 A XML-based feature modelling approach. ................................. 97

6.3 Generation of the application constraint checker. ....................... 98

6.4 A feature macro example. .......................................................... 99

6.5 The framework meta-model. ...................................................... 99

6.6 A framework model example. ..................................................... 100

6.7 Instance of the framework model. .............................................. 101

6.8 The constraint model. .............................................................. 102

6.9 The instantiation approach from the application feature model. .... 104

6.10 The structure of feature type information. .................................. 105

6.11 The graphical feature modelling tool. ...................................... 106

7.1 An overview of realized control applications. ............................ 112

7.2 The swing mass model (SMM). ................................................... 113

7.3 The ER1 robot. ................................................................. 113

7.4 The ER1 robot carrying a laptop. ............................................. 113

7.5 The RPX Lite board with PowerPC 823. .................................. 114

7.6 The A/D and D/A converter board. .......................................... 114

7.7 The personal computer connected to the SMM through the use of PCI A/D and D/A conversion board. .............................. 115

7.8 Jbed scheduling policy. .......................................................... 118

7.9 PERC scheduling policy. ........................................................ 124

7.10 The scheduling policy of the robot control application. .............. 128
7.11 The instantiated control application for the SMM deployed on the Jbed & PowerPC platform developed in the customized Bean Builder. 135

7.12 The instantiated control application for the ER1 robot deployed on the Java platform developed in the GIE. 137
List of Tables

2.1 The list of minor changes in the RTJ Framework ........................................... 30
2.2 Design patterns in the Unit Management Architecture ................................... 38
3.1 A list of shortcomings of standard Java for real-time applications ................. 51

7.1 Memory measurements of the SMM control application and “Hello World” program deployed on the Jbed & PowerPC platform ......................................................... 120
7.2 CPU measurements of the control application deployed on the Jbed & PowerPC platform ................................................................. 121
7.3 Memory measurements of the SMM control application and “Hello World” program deployed on the PERC & VxWorks & PC platform ................... 125
7.4 CPU measurements of the control application deployed on the PERC & Vx-Works & PC platform ................................................................. 126
7.5 Memory measurement of the robot control application and “Hello World” program deployed on the Java & PC platform .............................. 129
7.6 The differences between the test application deployed on a desktop and the target control application for the SMM deployed on the Jbed & PowerPC platform ................................................................. 130
7.7 The differences between the test application deployed on a desktop and the target control application for the SMM deployed on the PERC & VxWorks & PC platform. .......................... 132
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>AOCS</td>
<td>Attitude and Orbit Control System Framework</td>
</tr>
<tr>
<td>API</td>
<td>Application Program Interface</td>
</tr>
<tr>
<td>EDF</td>
<td>Earliest Deadline First</td>
</tr>
<tr>
<td>EMF</td>
<td>Eclipse Modelling Framework</td>
</tr>
<tr>
<td>FIFO</td>
<td>First In First Out</td>
</tr>
<tr>
<td>FTP</td>
<td>File Transfer Protocol</td>
</tr>
<tr>
<td>GC</td>
<td>Garbage Collection</td>
</tr>
<tr>
<td>GEF</td>
<td>Graphical Editing Framework</td>
</tr>
<tr>
<td>GGIE</td>
<td>Generic Graphical Instantiation Environment</td>
</tr>
<tr>
<td>HW</td>
<td>Hardware</td>
</tr>
<tr>
<td>JIT</td>
<td>Just-In-Time</td>
</tr>
<tr>
<td>JNI</td>
<td>Java Native Interface</td>
</tr>
<tr>
<td>JVM</td>
<td>Java Virtual Machine</td>
</tr>
<tr>
<td>NIST</td>
<td>National Institute of Standards and Technology</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>PC</td>
<td>Personal Computer</td>
</tr>
<tr>
<td>PDF</td>
<td>Portable Document Format</td>
</tr>
<tr>
<td>PS</td>
<td>Post Script</td>
</tr>
<tr>
<td>PVM</td>
<td>PERC Virtual Machine</td>
</tr>
<tr>
<td>RCC</td>
<td>Robot Control Center</td>
</tr>
<tr>
<td>RCM</td>
<td>Robot Control Module</td>
</tr>
<tr>
<td>RTF</td>
<td>Rich Text Format</td>
</tr>
<tr>
<td>RTJ</td>
<td>Real-Time Java</td>
</tr>
<tr>
<td>RTSJ</td>
<td>The Real-Time Specifications for Java</td>
</tr>
<tr>
<td>Acronym</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>--------------------------------------</td>
</tr>
<tr>
<td>RTOS</td>
<td>Real-Time Operating System</td>
</tr>
<tr>
<td>SGML</td>
<td>Standard Generalized Markup Language</td>
</tr>
<tr>
<td>SMM</td>
<td>Swing Mass Model</td>
</tr>
<tr>
<td>SW</td>
<td>Software</td>
</tr>
<tr>
<td>TBCC</td>
<td>Target Byte-Code Compiler</td>
</tr>
<tr>
<td>UMA</td>
<td>Unit Management Architecture</td>
</tr>
<tr>
<td>WORA</td>
<td>Write Once Run Anywhere</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
<tr>
<td>XSL</td>
<td>Extensible Stylesheet Language</td>
</tr>
</tbody>
</table>
Typographic Conventions

1. Important words are emphasized by *slanted* typeface.

2. Names of software products, technologies and programming languages are printed in *sans-serif* typeface.

3. Program sources are printed in *fixed-width* typeface.
CHAPTER 1

Introduction

The objective of this introductory chapter is to explain the motivation and goals of this thesis, provide an introduction in software framework design, implementation and instantiation. Furthermore, it outlines chapters’ organization.

Contents

1.1 Motivation .................................................. 13
1.2 Software Frameworks ........................................ 13
  1.2.1 Development Approaches for Software Frameworks ............ 14
  1.2.2 Software Framework for Control Systems ...................... 15
  1.2.3 How to Use a Software Framework? .......................... 16
1.3 Previous Work .............................................. 16
  1.3.1 Framework Design .......................................... 16
  1.3.2 Framework Implementation ................................. 17
1.1 Motivation

Software frameworks [42, 47, 67] are a software technology that promotes the reuse of entire architectures within a defined application domain. Software frameworks were successfully applied in control and robotics domains [13, 51, 66], where reusability of software assets across applications, quality and fast implementation play an important role. However, there is another fact that had influence on expanding development and usage of software frameworks, it is growing importance of software-related costs in the total development costs of software systems.

Software frameworks are rather new in the control software domain [8, 66], and there are two reasons for a slow acceptance of software frameworks in the control domain. First, control systems are a very conservative domain, even though some of the advanced software techniques were already adopted [39]. This is given mainly by the nature of the applications. Second, there is a widespread misconception that the control systems are simple applications, because they consist only of the implementation of control laws.

The work described in this thesis demonstrates that software frameworks can be used for the development of control systems even when they are implemented in the Java programming language [78]. A primary focus is on the instantiation process, the application development process from a software framework. The instantiation process is a complex task and usually is made manually. An important question is to whether we can automate the instantiation process and facilitate the development of applications based on a software framework.

Facilitation of the instantiation process is considered as the main motivation for the work presented in this thesis.

1.2 Software Frameworks

Software frameworks are a software technology that promotes the reuse of entire architectures within a narrowly defined application domain. An architecture is optimized for all applications within a certain domain and provides configurable components that support their instantiation. However, an architecture is not yet a framework and requires implementation and documentation describing proper use before being considered as one.

Use of framework technology is a means to manage software costs by increasing the level of software reuse. An alternative response to growing software costs in the control community is the adoption of tools like Matlab’s Real-Time Workshop [82] that allows a control system to be designed in a GUI-based environment and to have the code implementing the design automatically generated by the tool itself.
Software reuse has long been practiced by control engineers but mostly has been restricted to the code level: the reuse of individual routines or modules implementing recurring functions (e.g., matrix operations, common types of controllers, drivers for common units). However, reuse can take place at higher levels of abstraction. At architectural level, the software developer reuses a set of components defined by their external interfaces [72]. The external interfaces of a set of components define how the components do or can interact. To reuse architecture therefore means to reuse the components and their mutual interactions. Design patterns represent adaptable solutions to abstract design problems [47]. They encapsulate reusable design solutions and therefore allow reuse at the abstract design level.

Reuse can then occur at three levels: code reuse, architecture reuse, and design pattern reuse. Software frameworks can be seen as a means to reuse software at all three levels thus maximizing the benefits of reuse [66]. More specifically, a software framework captures the commonalities of applications in a target domain and encapsulates them in three sets of constructs:

- Domain-specific design patterns
- Abstract interfaces
- Concrete components

The domain patterns offer reusable design solutions to recurring design problems in the framework domain. The abstract interfaces define the points of adaptability where the framework can be adapted to match the needs of specific applications in its target domain. They also define the external interfaces of the framework components and hence define a reusable architecture. The concrete components implement some of the abstract interfaces and hence support the instantiation of the architecture.

1.2.1 Development Approaches for Software Frameworks

There are two main approaches for the development of software frameworks:

- Bottom-up
- Top-down

The bottom-up approach is a methodology where the framework is extended if an application does not fit to the domain covered by the framework. It is an incremental approach and usually should not be used, unless the framework is designed to be very small and extendable by user defined aspects.

The top-down approach starts with a solid analysis of a selected domain in order to capture all possible applications from the domain [27]. It leads to a fixed framework architecture,
1.2. Software Frameworks

which has a positive effect on framework reliability and consequently on reliability of the instantiated applications.

The software framework introduced in this thesis has followed the top-down approach. First, an analysis of the domain was made, then an appropriate software architecture has been designed and finally relevant components have been implemented.

1.2.2 Software Framework for Control Systems

Can framework technology be applied to control systems that operate under real-time constraints? In a sense the answer is trivially positive since real-time operating systems (RTOS) – a very successful instance of software reuse – are a form of framework. RTOSs are based on a small number of design patterns that deal with concurrency. Concrete instances of RTOSs impose a specific architecture upon their client applications (they define interfaces to which these applications must conform) and they provide reusable and application independent components that support the instantiation of that architecture.

Frameworks are primarily defined by how easily they can be adapted to match the needs of applications within their domain. Since design patterns offer encapsulation of adaptation mechanisms, they can be seen as the heart of a software framework. Hence, the first step in extending the framework approach to control systems is the development of a catalogue of design patterns for these types of systems. This requires the identification of recurring design problems in this field and the definition of abstract solutions for them. Some work has already been done in this area but it only covers concurrency and high-level architectural issues [31, 92]. The approach described in this thesis instead focused on design patterns addressing the software implementation, in particular the following functional aspects of control systems (see Figure 1.1):

- Management of control algorithms
- Management of failure detection checks
- Management of failure recovery actions
- Management of external units
- Management of manoeuvres
- Management of external commands
- Management of housekeeping data
- Management of operational mode

The applied design patterns are presented in [47, 66]. They provide a basis for the development of frameworks for control systems. However, two problems arise when they are instantiated in concrete architectures and components. Firstly, the architecture and the components must allow integration with standard OS or RTOS. Secondly, they must be compatible with real-time requirements. In order to explore these issues and to demonstrate the applicability of
1.2.3 How to Use a Software Framework?

A framework is typically implemented as a set of abstract interfaces and eventually completed by concrete components. Frameworks are not supposed to be stand-alone applications, but they allow creation of particular applications within their target domain. The process whereby an application is created by specializing a framework is called framework instantiation. Chapter 4 provides an introduction to framework instantiation problem.

1.3 Previous Work

1.3.1 Framework Design

Applications within the same domain tend to share the same type of requirements. For instance, in the control system domain we can find a requirement for sensor measurements, control algorithms, failure detection and recovery actions, and so on. The particular algorithms that are used vary from application to application, but the general structure remains the same. The common and variable properties from our domain that are required to be covered by a software architecture are given in [42, 47, 67].

A domain is defined in a process known as domain analysis. The main purpose of the domain analysis is to collect the relevant domain information and integrate them into a domain model [27]. An overview of domain analysis methods that shape the domain model are discussed in [4, 27, 56, 77].

An appropriate software architecture is developed on the basis of a domain model. The development of a software architecture has to take into account requirements on functionality,
robustness, performance, adaptability, extendability, reusability and maintenance [15, 27]. The architecture of software frameworks is one of feasible technologies that capture variability of application dependent parts and reusability of common parts and which also covers the mentioned requirements on a software architecture. Object-oriented software frameworks have been chosen for the domain of control systems in [8, 66]. There are also software frameworks that are focusing on domains close to control domain. Consider for example the robotics domain, where there is a flexible software framework for real-time motion robot control called OROCOS [12, 13]. This is a software framework for robot motion planning, coordinate transformation and control which is implemented in Java [51]. Another example is an agent based software framework for multi-sensor, multi-actuator systems for an autonomous intelligent robot [87].

1.3.2 Framework Implementation

This thesis refers to the Java implementation of the RTJ Framework [8, 20]. In recent years, many have proposed that the use of Java be extended to real-time systems, including control systems. The theoretical bases are discussed in several publications [10, 11, 59]. One of the objectives to develop the RTJ Framework was to test the maturity of real-time Java implementations on a concrete instance. The Java language [78] is a natural choice for implementing software frameworks. It is fully object-oriented and offers a construct to represent abstract interfaces that play a crucial role at the framework design. Additionally, it offers built-in multitasking and networking support and its wide user base leads to lower development costs. The presence of automatic documentation facilities is another asset of Java [37], especially in domains like control systems – where there are demanding documentation requirements. The use of Java for a real-time control system places constraints on the application code and on the run-time system. Both must guarantee predictability of timing behavior, which could be achieved by constraints on the application code. The constraints on the run-time system primarily relate to the enforcement of deterministic schedulable policies, garbage collection and synchronization mechanisms [53, 83]. Such constraints are not mandated by the Java language specifications but a number of suppliers provide Java run-times that claim to be real-time compliant [1, 28, 38, 40, 60, 84].

1.3.3 Framework Instantiation

The problem of framework instantiation has been the object of research for several years. Some solutions [30, 42, 43, 44, 63] relied on putting together a body of rules (also known as "recipes") to aid the developer in using the framework. More advanced versions of this approach use agents to assist the framework instantiation process [64] but most current work looks at generative techniques [27] as a means to automate the framework instantiation process [16, 25, 26, 45, 46, 62]. Such techniques rely on domain-specific languages (DSL) to specify the target application. Although effective and powerful, these techniques tend to be comparatively complex and present a high barrier to entry for general users. The distinctive feature of instantiation approaches proposed in this thesis is their simplicity and reliance on mainstream
technologies and tools that have the potential of bringing it within the reach of non-specialist users.

1.3.4 Basis Provided by the Research Group

The need for a concept that would facilitate the framework instantiation process was expressed in [64, 66]. The work presented in this thesis started after establishing the collaboration between the Automatic Control Laboratory of the ETH in Zürich and the Department of Computer Science of the University of Constance, where at that time a group of Prof. Wolfgang Pree was located dealing with framework technology and developing a software framework for Attitude and Orbit Control Systems (AOCS Framework) [66].

1.4 Goal of This Work

The overall goal of this work is to develop a generic concept for an instantiation of object-oriented software frameworks.

The main objectives are:

• $O_1$: To re-engineer the AOCS Framework to an object-oriented software framework for general control systems.
• $O_2$: To evaluate suitability of object-oriented programming language Java for real-time programming.
• $O_3$: To develop new concepts that facilitate a framework instantiation process.
• $O_4$: To demonstrate the new instantiation concept on concrete examples.

1.5 Chapter Layout

The thesis includes eight chapters, where four of them called “Design”, “Implementation”, “Automated Instantiation”, and “Feature-Based Modelling and Instantiation” contain the most important achievements that are later demonstrated on particular problems described in the chapter dedicated to test cases.

Chapter 1 is an introductory chapter providing an overview on the software framework and other key technologies used for solving various problems described in the thesis. It also provides a discussion about other related approaches.

Chapter 2 explains design principles that are necessary for the development of software frameworks that should be compliant with generative programming techniques. It highlights the main differences between the RTJ Framework and AOCS Framework.
Chapter 3 presents the state-of-art of real-time Java and discusses the suitability of Java for real-time control applications. It also reports experiences with an implementation of the RTJ Framework to standard Java and two commercial real-time versions of Java.

Chapter 4 provides an introduction to a framework instantiation problem, and an overview of available and new instantiation techniques.

Chapter 5 introduces a new automated instantiation concept based on generative techniques. The instantiation process is automated in the sense that designers configure and assemble the framework components using intuitive visual operations in a GUI-based environment, with the configuration actions automatically transformed to framework instantiation code.

Chapter 6 introduces a new instantiation concept using feature modelling technique. The developed concept is generic in the sense of supporting different frameworks. It allows both the modelling of the software framework and of the applications instantiated from it.

Chapter 7 demonstrates and compares the traditional and the new automated and feature-based framework instantiation concept on a laboratory swing mass model (SMM) and a ER1 robot system [41]. Although both the SMM and ER1 robot are rather simple systems, the target applications based on the RTJ Framework allowed us to demonstrate all typical functionalities that can be identified in control systems.

Chapter 8 highlights main results and contributions and concludes the thesis with the overview of further prospective research work.
The first and essential steps in the software framework development process are a domain analysis and design. A proper software framework design can significantly simplify an instantiation process. This chapter provides framework design principles that facilitate the framework instantiation process.

Contents

2.1 Introduction ......................................................... 23
2.2 Motivation .......................................................... 24
2.3 Introduction to Domain and Application Engineering .......... 24
2.4 Domain and Application Engineering for RTJ Framework ....... 25
  2.4.1 Framework Development Process ............................ 25
  2.4.2 Application Development Process ............................ 26
2.5 RTJ Framework versus AOCS Framework ........................ 28
  2.5.1 Target Domain ................................................. 28
  2.5.2 Design Differences ........................................... 28
2.6 Framework Design Principles in the RTJ Framework ........... 31
  2.6.1 Naming Conventions ......................................... 31
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.6.2 Separation of Instantiation and Application Methods</td>
<td>33</td>
</tr>
<tr>
<td>2.6.3 Independence of Instantiation Methods</td>
<td>34</td>
</tr>
<tr>
<td>2.6.4 Avoidance of Static Properties and Methods</td>
<td>34</td>
</tr>
<tr>
<td>2.7 Design Examples</td>
<td>35</td>
</tr>
<tr>
<td>2.7.1 Unit Management Architecture for RTJ Framework</td>
<td>35</td>
</tr>
<tr>
<td>2.7.2 Matlab Wrapper for RTJ Framework</td>
<td>43</td>
</tr>
<tr>
<td>2.8 Conclusions</td>
<td>45</td>
</tr>
</tbody>
</table>
2.1 Introduction

Design process for software frameworks is not standardized. Throughout this text, the investigation relate to the **RTJ Framework**. The RTJ Framework is component-based and object-oriented. It uses *inheritance* and *object composition* as adaptation mechanisms (see Figure 2.1 and Figure 2.2). The first mechanism is used for *static behavior adaptation*, whereas the second one for *dynamic behavior adaptation*. An important property of these adaptation mechanisms is that they allow the behavior of a target class to be adapted without touching its source code. The RTJ Framework architecture is captured by a set of design patterns [47, 66, 67]. The design patterns are the vehicle through which the architecture pre-defined by the framework is captured.

The RTJ Framework is formally described by a *feature model* [54] that captures the commonalities and the variability of applications within the framework domain. Furthermore by a UML class model [61] that describes its architecture, and by a detailed documentation of the framework classes. The RTJ Framework was developed by means of domain engineering techniques, and is compliant with domain engineering process.

![Figure 2.1: The adaptation mechanism based on inheritance.](image-url)
2.2 Motivation

Software frameworks provide a generic architectural skeleton from which applications within a specific domain can be rapidly instantiated. The first objective to develop the RTJ Framework is to provide a design and implementation that is not specific to any particular area in the control domain, but rather can be easily customized for any domain. The second objective is to prove suitability of software frameworks for the domain of control systems. The third objective is to provide a testing environment for new concepts facilitating the instantiation process.

2.3 Introduction to Domain and Application Engineering

Domain engineering aims at the development of reusable software, for example, a generic system whose parts can be instantiated and reused in other systems. Domain engineering is applicable for development of software frameworks, components, domain-specific languages, and generators [19, 27].

Domain engineering addresses the systematic creation of domain models and architectures. It covers all the activities for building software core assets. The domain engineering process can be divided into three phases (see Figure 2.3):

**Domain Analysis** – identifies a domain, and captures the variation within a domain. The
output is a formal domain model.

**Domain Design** – constructs an adaptable and reusable domain architecture. The output is a design model.

**Domain Implementation** – is a phase whereby the design model is implemented. It also defines the mechanisms for translating requirements into systems created from reusable components.

Domain engineering also supports *application engineering*, which uses the models and architectures to build particular systems. Domain and application engineering are complementary processes. The objective of domain engineering is to cover certain domain, whereas the objective of application engineering is a development process of one particular system. The application engineering process can be divided again into three phases (see Figure 2.3):

**Requirements Engineering** – is a process where an application engineer use the domain model and the customer needs to define a list of new requirements and features for the required application.

**Design Analysis** – compares the requirements defined in requirement engineering phase and produces product configuration data, and for requirements not compliant with the existing assets, it creates new component specifications.

**Integration & Test** – makes a composition of building blocks to the needs of a target application, and also performs validation tests.

The extended description of domain and application engineering processes can be found in [19, 27, 50, 54].

### 2.4 Domain and Application Engineering for RTJ Framework

In the context of domain and application engineering processes the framework creation corresponds to the domain engineering process, whereas creating of framework applications, i.e. framework instantiation, corresponds to the application engineering process. Figure 2.4 shows the generic development process shown in Figure 2.3 adapted to the software framework and framework instantiation process.

#### 2.4.1 Framework Development Process

The *framework development process* can be divided into three phases:
Domain Analysis – identifies and characterizes the framework target domain. The output is a formal domain model. For the RTJ Framework, the output of the domain analysis phase is a formal model of the target domain expressed as a feature model [54].

Domain Design – constructs an adaptable and reusable domain framework architecture based on design patterns. There are three outputs at two levels of abstractions. At the abstract design level, the framework design is captured by a set of design patterns that propose design solutions [66]. The design patterns encapsulate reusable architectural solutions for applications within the framework domain [42, 47, 67]. At the architectural design level, the framework is described by a UML class diagram that covers all its concrete and abstract classes [61].

Domain Implementation – is a phase whereby the design model is implemented. The RTJ Framework is implemented as a set of Java classes. Some of the Java classes are concrete and implement the framework concrete components, others are abstract and implement the framework abstract interfaces.

2.4.2 Application Development Process

The application development process, framework instantiation, is divided into three phases:
2.4. Domain and Application Engineering for RTJ Framework

Requirements Engineering – is a phase where a model of the target application is constructed. The application model is constructed as an instance of the framework model. The application model is for RTJ Framework based applications expressed as an XML document. The RTJ Framework provides an XML Schema (XSD) that defines the structure of this document. The definition of the document itself must be done by the application designer on the basis of the application requirements.

Component Tailoring – adapts components provided by the framework to the needs of the target application. The outputs of this phase are application-specific components and a configuration knowledge for the target application. The RTJ Framework supports the following adaptation techniques for component tailoring to the needs of a target application:

- Component Configuration: the components provided by the RTJ Framework or created by extending its concrete and abstract classes, are intended to be configured during the application configuration phase. Component configuration consists of setting the values of the component attributes. Since some component attributes are a type of class, this includes the object composition adaptation mechanism.
• Class Extension: the RTJ Framework provides a number of Java classes that can be adapted by extension. This includes both the override of existing methods, the definition of virtual methods, and the addition of new methods.

• Automatic Code Generation: the RTJ Framework offers a generator that can generate new application-specific classes from a high-level specification provided by the designer. The classes thus generated are guaranteed to be plug-in compatible with other RTJ Framework classes.

Component Composition – is a phase where the application-specific components are composed with each other to construct the target application. The RTJ Framework supports the component composition phase. This is the phase where the individual components adapted to the specific needs of a target application are composed together to construct the final application executable. Component composition can be performed manually, but the new approaches introduced in this thesis automate the instantiation process.

2.5 RTJ Framework versus AOCS Framework

The RTJ Framework design is influenced by the AOCS Framework design. However, the main difference lies at the implementation level. The RTJ Framework is implemented in Java (see Chapter 3) whereas the AOCS Framework is implemented in C++.

2.5.1 Target Domain

The target domain of both frameworks are control software systems. In particular, the AOCS Framework is an object-oriented framework for Attitude and Orbit Control Systems of satellites [66]. The RTJ Framework covers the AOCS domain as well, however one of the RTJ Framework’s objective is to provide a design and implementation that is not domain specific, but rather can be easily customized for a particular domain.

2.5.2 Design Differences

The RTJ Framework and AOCS Framework consist of three types of constructs:

• A set of domain-specific design patterns
• A set of abstract interfaces
• A set of concrete components

One of the RTJ Framework objectives is to maintain consistency with the AOCS Framework at the level of design, architecture, and implementation. Therefore, the rules applied in the design of the RTJ Framework are as follows:

• Design Patterns – Framework design patterns are the patterns introduced in [47, 66, 67].
2.5. RTJ Framework versus AOCS Framework

- **Abstract Interfaces** – If it is possible then methods in abstract interfaces maintain the same name, the same number and types of methods with the only difference that C++ pointers are changed to Java references.

- **Components** – Components are using the same rules as for the abstract interfaces.

These rules could not be followed in all cases. Differences between C++ and Java, uncertainties about the real-time Java specifications, and a desire to take advantage of the JavaBeans standard [36] made it sometimes necessary to deviate from the original C++ design and implementation of the framework.

2.5.2.1 Minor Changes

Minor changes have been undertaken: this is generally to fix bugs in the original AOCS Framework, and ensure commonality of naming conventions (see Section 2.6). Minor changes are the changes that are localized to specific interfaces, classes or methods in the framework, the most important minor changes are listed in Table 2.1.

2.5.2.2 Major Changes

**EVENT MODEL**

The RTJ Framework introduces a new event model. The objective was to implement the framework event mechanism that is compatible with the JavaBeans event model. Compatibility was achieved at the syntactical and at the naming level. This approach should allow use of standard component composition environments to manipulate the framework components. Nevertheless, the difference between the Java event model and the one designed for the RTJ Framework is fundamental. Applications using the standard Java event model create an event dynamically before firing it, whereas in the applications instantiated from the RTJ Framework cannot do so as dynamic creation of objects is not allowed. The event model for the RTJ Framework works as follows. When they need to fire an event, they load the event parameters into the dummy event and pass the latter to the `fireEvent` method. The event listener begins processing the incoming event by copying its parameters into an internal data structure. This releases the dummy event in the event source for use in the next event firing operation. In order to avoid data corruption in a multithreading environment, the copying of the event parameters to the internal data structure is done in a synchronized section.

There is one more important different between the AOCS Framework and RTJ Framework event model. The AOCS Framework events are stored in event repositories, each event source could fire events to only one event repository. In the RTJ Framework, instead, event sources maintain an array of event listeners and can therefore potentially send their events to multiple repositories.
<table>
<thead>
<tr>
<th>AOCS Classes &amp; Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runnable</td>
<td>Class name was changed to ActiveObject to avoid conflict with Java Runnable class.</td>
</tr>
<tr>
<td>AbstractControlChannel</td>
<td>Class implements the interfaces DataSource and DataSink.</td>
</tr>
<tr>
<td>ControlChannelBlock</td>
<td>Class implements all inherited abstract methods from the class AbstractControlChannel and implements methods defined in interfaces DataSource and DataSink.</td>
</tr>
<tr>
<td>ControlChannelSuperBlock</td>
<td>Class implements all inherited abstract methods from the class AbstractControlChannel and implements methods defined in interfaces DataSource and DataSink.</td>
</tr>
<tr>
<td>DataltemRead</td>
<td>DataltemRead objects encapsulate a reference to the data source. This mechanism is used internally.</td>
</tr>
<tr>
<td>DataltemWrite</td>
<td>DataltemWrite objects encapsulate a reference to the data sink. This mechanism is used internally.</td>
</tr>
<tr>
<td>*.allocateMemory</td>
<td>The initialization methods in components are changed to setter method in the spirit of the JavaBeans architecture. For instance, method ObjectList.allocateMemory is renamed to setSize. Similarly, method SystemManager.initialize is replaced by setter methods for each of its parameters.</td>
</tr>
<tr>
<td>*.initialize</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1: The list of minor changes in the RTJ Framework.

Linking of Data Flow Components

The RTJ Framework introduces a new concept for linking of data flow components, which is based on data source and sink mechanisms.

- A data source is an object with a fixed number of outputs that participates in a data flow. Data sources offer read-only access to their outputs.
- A data sink is an object with a fixed number of inputs that participates in a data flow. Data sinks offer read-write access to their inputs.

Data flow components see each other as instances of type DataSink and DataSource.
In the AOCS Framework, data flow components were linked to their data sources and data sinks through DataItemRead and DataItemWrite objects. These objects encapsulate a pointer to the data source or data sink. This mechanism is not possible in Java because in Java pointers to data are not explicitly allowed.

UNIT MANAGEMENT ARCHITECTURE

The Unit Management Architecture (UMA) is a reusable architecture for operation and maintenance of external devices based on domain-specific design patterns, a set of interfaces, abstract classes, and customized components. In comparison to the UMA that was already present in the AOCS Framework [66], the new UMA has been improved with a general abstract unit model, mechanism of execution of external devices and includes a new mechanism of handling of smart units. More detailed discussion can be found in Section 2.7.1.

MATLAB WRAPPER

Embedded control systems always include control algorithms developed by control engineers. Control engineers are used to design control algorithms in the Matlab and Simulink computation and simulation environment. The RTJ Framework introduces a new concept for integration algorithms developed in the Matlab/Simulink environment called Matlab Wrapper (see Section 2.7.2). Matlab Wrapper encapsulates the algorithm and makes it directly pluggable into an application instantiated from the RTJ Framework.

2.6 Framework Design Principles in the RTJ Framework

Frameworks are designated to be instantiated. The proposed framework design principles applied on the RTJ Framework already take into account the instantiation friendly process, which could be automated. However, it is necessary to stress that the rules given below should not be taken literally, because probably not all rules will be appropriate for all frameworks or sometimes it will not be possible to implement them all.

2.6.1 Naming Conventions

Naming conventions play an important role for the framework instantiation process. Naming conventions are applicable at different levels of the framework code, at the level of packages, classes and interfaces, properties and methods. The automated framework instantiation processes (described in Chapter 5), which is directed by generative programming techniques, relies mostly on naming conventions of properties and methods defined in classes and interfaces. Component properties and methods should be constructed so that names of application and instantiation operations are easily distinguishable, and that the instantiation operations conform to predefined naming patterns that allow the semantics of an operation to be inferred from its name.
Manipulation of the framework components within an automated instantiation environment requires the environment to find out information about the components. Since the components are manipulated only through the methods they expose, the information the environment needs only concerns these methods. Use of naming patterns for the methods is a simple way through which this information can be encoded. The alternative is a look-up table which the environment can query to find out information about the component methods.

The naming conventions used in the RTJ Framework adopted Java or more precisely JavaBeans naming conventions. One of the main reasons was assumed easier integration of framework components into component composition environments supporting JavaBeans component standard [36].

The most important naming conventions applied in the RTJ Framework are listed in the following paragraphs.

### 2.6.1.1 Property Access Methods

The methods used for setting and getting property values conform the property naming pattern:

```java
public void set<PropertyName>(<PropertyType> value);
public <PropertyType> get<PropertyName>();
```

In the case of a boolean type of the property, the getter method can be formulated as:

```java
public boolean is<PropertyName>();
```

### 2.6.1.2 Indexed Property Access Methods

The methods used for setting and getting indexed properties conform the indexed property naming pattern:

```java
public void set<PropertyName>(<PropertyType>[] value);
public <PropertyType>[] get<PropertyName>();
```

These methods are accessing the entire array of property values. The methods allowing an access to particular values are:
2.6. Framework Design Principles in the RTJ Framework

```java
public void set<PropertyName>(int index,<PropertyType> value);
public <PropertyType> get<PropertyName>(int index);
```

### 2.6.1.3 Container Property Access Methods

The methods used for adding and removing objects from a property that acts as an object container conform the container property naming pattern:

```java
public void addObject(<PropertyType> value);
public void removeObject(<PropertyType> value);
```

or

```java
public void add<PropertyType>(<PropertyType> value);
public void remove<PropertyType>(<PropertyType> value);
```

### 2.6.1.4 Registering for Event Notification

The methods used for registration and removing of event notifications conform the event-listener naming pattern:

```java
public void add<ListenerType>(<ListenerType> listener);
public void remove<ListenerType>(<ListenerType> listener);
```

### 2.6.2 Separation of Instantiation and Application Methods

One of the framework design objectives is to provide specific methods for the instantiation process and methods implementing core framework functionalities that are executed during the application operation phase.

The splitting of methods participating in the instantiation process and methods used during the operational phase of applications is the key concept used in the automated instantiation environment. The idea is to replace complex components with proxy components that contain only instantiation methods and therefore are easier to manipulate in a specialized instantiation environment. The simplest way to ensure that this split is possible is to properly design
the framework components in such a way that the methods can be separated between those used only during the instantiation process, and those used only during the operational phases of applications. It appears that the best way to achieve this type of separation is to gather all instantiation relevant methods in dedicated abstract interfaces.

### 2.6.3 Independence of Instantiation Methods

The other framework design objective should focus on making the instantiation methods as independent from each other as possible.

The instantiation process is a sequence of operations performed upon the framework components. In general, the instantiation sequence has to satisfy some ordering constraints. For example, some operations can only be performed after some other operations have been performed; or the way some components are configured may be dependent on the outcome of previous configuration actions. These types of dependencies have to be minimized.

In the case of the RTJ Framework, one particular source of dependencies among instantiation operations is the use of dynamic memory allocation. Some framework components internally use data structures whose memory must be allocated at configuration time. Typically, these components expose a property called size that defines the size of the internal data structure. A call to the property setter method causes memory allocation for this data structure. Since some other configuration actions use this data structure, it is necessary that the size property be set before such configuration actions are executed. The situation is demonstrated on the following code sample:

```java
    tmList.setListSize(10);       // memory allocation
    tmList.addObject(jbedClock);  // data structure configuration
    tmList.addObject(configRep);   // data structure configuration
    tmList.addObject(failureRep);  // data structure configuration
    ...
```

### 2.6.4 Avoidance of Static Properties and Methods

The next recommendation states that the framework should be designed so that the component properties and methods are not static (in the sense of having static storage).

One of the requirements of an instantiation environment is that it allows the functionalities of the application to be tested during the application configuration process. This is very useful, because the designer can check the correctness of his configuration actions even for an incomplete application. In the case of the customized instantiation environment (see Section 5.2) this is done by switching the environment from “design mode” to “run mode”. Every time there is
a switch into "run mode", the target application is dynamically reconstructed in order to implement all the configuration actions defined in the instantiation environment up to that moment by the designer. It is important to stress that the application must be reconstructed newly every time there is a request for the transition into "run mode". This is because there is no guarantee that it is possible to bring the application from the state it had at the previous transition into "run mode" to the state that is required at the latest transition into "run mode". Hence, the safest course is always to load and decode all the configuration actions defined by the application designer and to use them to build a new image of the target application. When the system is brought again into "design mode" the application is destroyed (i.e. all its components are allowed to be reclaimed by the garbage collector). All configuration information should be destroyed with it to avoid the next reconstruction of the application being contaminated by the previous one. This however is difficult to do in the presence of static properties in the application components because the setting of static properties remains even after the application is destroyed. The RTJ Framework has a small number of static properties which required special treatment in the instantiation environment. This added unnecessary complexity to the environment which could have been avoided if this recommendation would be followed. It is important to note that other instantiation environments may use different mechanisms to simulate execution of the application during the configuration process but in most cases they will need to create and destroy the target application dynamically and will want to ensure that there is as little coupling between different instantiations as is possible.

2.7 Design Examples

Two reusable solutions, the Unit Management Architecture and Matlab Wrapper, were selected to demonstrate the development process of reusable architectures that together with other parts form the RTJ Framework.

2.7.1 Unit Management Architecture for RTJ Framework

The Unit Management Architecture (UMA) is a reusable architecture for operation and maintenance of external devices based on domain-specific design patterns, a set of interfaces, abstract classes, and customized components. The UMA is a part of the RTJ Framework. In comparison to the UMA version of the AOCS Framework [66], the UMA presented here has improved abstract unit models, mechanisms of execution of external devices, and newly includes a mechanism of handling of smart units.

2.7.1.1 Domain Description

The main objective of the UMA is to handle a wide spectrum of external units used by embedded control applications. They can be divided into real units, and fictitious units. Another objective is to provide other advanced features such as a reconfiguration of redundant units or mode dependency.
REAL UNITS
The real units can be further divided into basic and smart units. The basic units are sensors and actuators that are processing only ordinary data types. The smart units have the same functionalities as basic units, it means that they process ordinary input and output data, but in addition to that they are able to process special commands, such as reset or self-test, etc.

FICTITIOUS UNITS
The concept of fictitious units is used for combinations of components that process unit data. A typical example is a data filter shown in Figure 2.5.

![Sensor Filter Controller](real unit fictitious unit final user)

Figure 2.5: The connection of a real unit, fictitious unit and final user.

RECONFIGURATION OF REDUNDANT UNITS
There are situations, mainly in critical applications, when some parts of hardware have back-up substitutions. The UMA has to provide a software solution for a case when an unit does not respond and therefore needs to be replaced by an identical back-up unit, e.g. switching of sensors.

MODE DEPENDENCE
The UMA has to provide a functionality to operate under different modes.

2.7.1.2 Domain Analysis – Feature Model
Figure 2.6 presents the feature model of UMA. The top feature “RTJ Framework” has one sub-feature “External Unit Set”. In reality the “RTJ Framework” feature has more than one sub-feature, their number is adequate to a number of functionalities provided by the framework.

The feature model diagram shown in Figure 2.6 states that:

- “External Unit Set” may provide a facility to listen to unit events (the optional feature “Listen to Unit Event”).
- “External Unit Set” can operate under different modes (the optional feature “Mode Dependence”).
- “External Unit Set” can be a member of reconfiguration group (the optional feature “Member of Reconfiguration Group”).
- “External Unit Set” has at least one “Unit”, which is either “Sensor” or “Actuator”, and can have some smart functions (the optional feature “Smart Functions”).
2.7. Design Examples

- "Sensor" and "Actuator" must be further specified, a sensor and actuator for Jbed platform [38] is available or a custom one can be defined.
- "Sensor" may have storing facility for measured data, it is expressed by the optional feature "Storable".
- "Storable" feature is in fact feature macro. It means that "Storable" can appear at other places of the feature model (feature macro mechanism is described in Section 6.2.2.2).

Figure 2.6: The feature model of the Unit Management Architecture.
2.7.1.3 Design

**Design Patterns in the Unit Management Architecture**

The design of the UMA is based on three design patterns, namely on Command, Proxy, and Strategy. In general the Command design pattern encapsulates a request as an object. The Command design pattern is a general solution suitable for the handling of commands that are not known in advance for smart units that are also not known. The second design pattern used in the unit management architecture is the Proxy design pattern that controls an access to an object. In the context of the unit management architecture the proxy is the unit model that acts as a stand-in for the real external unit. The third design pattern, the Strategy design pattern, appears on several places in the unit management architecture. Its task is to make the unit management architecture flexible and reusable.

<table>
<thead>
<tr>
<th>Design Patterns</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command</td>
<td>Provide a solution for handling of general commands executed by smart units.</td>
</tr>
<tr>
<td>Proxy</td>
<td>Act as stand-in for the real external unit.</td>
</tr>
<tr>
<td>Strategy</td>
<td>Define a family of customized objects and make them interchangeable.</td>
</tr>
</tbody>
</table>

Table 2.2: Design patterns in the Unit Management Architecture.

The design patterns used in the UMA are based on the patterns introduced in [47], although they have been customized. Participants of the customized Command design pattern are an abstract interface `UnitCommand`, concrete commands defining operations executed by smart units, a concrete unit that behaves as a receiver of concrete commands and that knows how to perform the operations encapsulated in a concrete command, and finally the `UnitManager` that represents the invoker of the commands. The UML notation of the customized Command design pattern is shown in Figure 2.7.

The abstract unit model defines an interface for Proxy representing the real unit. The UML notation of the customized Proxy design pattern is shown in Figure 2.8.

The Strategy design pattern has been applied in several parts of the UMA. For instance, the `UnitManager` component is responsible (besides of others tasks) for handling of units in accordance with a trigger strategy that is interchangeable. The UML notation of the customized Strategy design pattern is shown in Figure 2.9.

Design patterns consist of interfaces, abstract classes and classes that can implement or extend them. The UMA provides four interfaces, two abstract classes and four components.
Figure 2.7: The customized Command design pattern.

Figure 2.8: The customized Proxy design pattern.
INTERFACES IN THE UNIT MANAGEMENT ARCHITECTURE

UnitFunctional is an interface for objects representing the functional exchanges between the embedded control application and an external unit (either real or fictitious). A component representing an external unit that provides a functional data exchange facility has to implement the UnitFunctional interface. Such a component can then be easily plugged into the entire framework architecture.

UnitHousekeeping is an interface for objects representing the housekeeping exchanges between the embedded control software and an external unit (either real or fictitious). This interface is developed for units that in addition to functional data, provide data about their internal state, like a status of a self-test, an operational temperature, etc.

UnitModeManager is the interface defining the ability to operate under different modes. This interface is characterizing the mode manager for the unit managers. The mode manager manages one single strategy representing the "triggering strategy" for the unit manager. A triggering strategy defines a set of units and the operations to be performed upon them.

UnitCommand interface provides a general approach for implementing commands of external smart units. A software component representing a smart unit that is able to process a command
has to implement all methods of the UnitCommand interface.

Let use methods declared in the UnitCommand interface for a demonstration of the naming convention concept and the concept of separation of instantiation and application methods proposed in the section describing the framework design principles (see Section 2.6).

The UnitCommand interface contains three methods:

```java
/**
 * A call of this method causes the actions associated with
 * the command to be executed.
 */
void execute();

/**
 * Set a value of the iteration flag.
 */
public void setIterationFlag(boolean iFlag);

/**
 * Return a value of the iteration flag. The value is false
 * if the command is executed only once, true value if is
 * executed at each UnitManager invocation.
 */
public boolean getIterationFlag();
```

Based on the proposed naming conventions, it is evident that the method `execute()` is the application method, whereas the method `setIterationFlag(boolean iFlag)` participates only during the instantiation process. The third method `getIterationFlag()` that retrieves a property value of type boolean is the instantiation method as well, however there could be a situation when this method will be called during the operational mode of the application.

### Abstract Classes in the Unit Management Architecture

Data exchange and command execution are the main tasks of the framework. The concept of data and command exchange is captured in a general abstract unit model and implemented in class `Unit`. The concept implemented in the `Unit` class is sufficient to handle both types of units: basic and smart. The basic unit is in general a simple single or multi input and output external device that is able to acquire data or perform some action based on received data. A smart unit is a single or multi input and output device that is performing actions according to
received special commands.

The unit abstract model provides a buffer for input raw data, a buffer for output raw data, and a buffer for housekeeping raw data. The buffers store acquired data from an external hardware or data that is ready to be sent to an external hardware. The raw data stored in the buffers are usually transformed to a data range that is used by the application. The unit abstract model provides methods to plug-in data converters by calling dedicated methods, and creates links with destination buffers and data pools. The exact structure of the unit abstract model is shown in Figure 2.10.

![Figure 2.10: The Unit abstract model.](image)

UnitTriggerStrategy provides a list of references to UnitFunctional or UnitHousekeeping objects and abstract operations for data acquisition, data sending or synchronization that need to be performed upon them.

**COMPONENTS IN THE UNIT MANAGEMENT ARCHITECTURE**
BasicUnitReconfigurer component handles reconfigurations across a set of several identical units.

DefaultUnitTriggerStrategy is the default unit trigger strategy component that provides a complete implementation of the functional and housekeeping data transfer between the unit hardware and the application.

FollowerUnitStrategyModeManager defines particular mode change strategy.

UnitManager performs functional and housekeeping actions upon unit proxies (upon the hardware).

### 2.7.2 Matlab Wrapper for RTJ Framework

The Matlab Wrapper is an abstract class that acts as a general container for code which has been automatically generated by Real-Time Workshop that is a part of the Matlab environment. The Matlab Wrapper approach was developed for a simple embedding of automatic generated control algorithms from Simulink/Real-Time Workshop into a specific RTJ Framework component. The general Matlab Wrapper and algorithm specific Matlab Wrappers are a part of the RTJ Framework; they are located in the SequentialData-Processing package of the RTJ Framework.

#### 2.7.2.1 Domain Description

The community of control engineers use Simulink and Real-Time Workshop to design and implement control algorithms. The objective is to reuse these algorithms and integrate them into the RTJ Framework where will be available for instantiation process.

#### 2.7.2.2 Design

The design of Matlab Wrapper is based on Unification design pattern [47]. The Unification design pattern applied for the problem of wrapping C code generated by Real-Time Workshop consists of MatlabBlock abstract class that is supposed to be extended and implemented by classes wrapping particular algorithms. For instance, MatlabGainBlock and MatlabIntegratorBlock classes, which extend MatlabBlock abstract class, act as wrappers for a gain and integrator block from the Simulink library (see Figure 2.11). The UML notation of the customized Unification design pattern is shown in Figure 2.12.

#### 2.7.2.3 Integration of C Code

The architecture of Matlab Wrapper is generic and therefore can be implemented by any object-oriented language. Since the RTJ Framework was implemented in Java, a problem of
Figure 2.11: The gain block from the Simulink library.

C code integration into Java implementation of the framework had to be solved. Fortunately, Java platform supports legacy code integration. The technology allowing C/C++ and assembly code integration in an application written in Java is called Java Native Interface (JNI).

The RTJ Framework was designed and implemented with respect to real-time requirements of instantiated applications. The important issue is to ensure that instantiated applications still...
can achieve real-time constraints even when legacy code substitute a certain part of application functionality. This task, however, belongs to providers of real-time Java platforms.

## 2.8 Conclusions

Several general conclusions can be drawn from an experience with software framework design.

The first concern relates to a framework design. A proper software framework design can significantly simplify instantiation process that ideally should be done automatically, but also manually. On the other hand, it will always be possible to automatize the instantiation process for almost any reasonably design framework with smaller or bigger effort. If it is known in advance what instantiation procedure will be used, then the framework design should be developed according to it. Based on practical experiences it can be stated that the framework design principles suggested in this chapter (in particular see Section 2.6) leads to a framework design that is instantiation friendly.

A software framework is ideally suited to modelling the non-algorithmic part of the control software and for this reason Matlab based approach can be seen as a complementary technique. Both approaches aim to bring down software costs. Matlab excels at modelling control algorithms, whereas a control system mostly requires functions like unit management, command processing, housekeeping data management, failure detection and failure recovery, and other functions for which Matlab does not provide specific abstractions and which are unable to model effectively in the Matlab environment.
Current control systems are typically implemented in C or Ada. The motivation for switching to Java is to make use of the greater safety and multitasking features and more recently also a support for hard real-time operations. This chapter focuses on proving the maturity and suitability of real-time Java for the software framework technology covering the control domain.

Contents

3.1 Introduction .................................................. 49
3.2 Motivation .................................................... 49
3.3 Standard Java and Real-Time Java ......................... 49
   3.3.1 Approaches to Real-Time Java ....................... 50
3.4 Implementation Aspects of the RTJ Framework ............ 54
   3.4.1 Framework Components as JavaBeans ............... 55
   3.4.2 Real-Time Issues .................................. 55
   3.4.3 Scheduling and Synchronization .................... 56
   3.4.4 Framework Memory Model ......................... 57
   3.4.5 Integration of Non-Standard Components ............ 58
Chapter 3. Implementation

3.4.6 Testing ............................................. 59
3.4.7 Documentation Approach .......................... 62
3.5 Conclusions ............................................ 62
3.1 Introduction

Object-oriented software frameworks can be implemented in any object-oriented language, which includes the mainstream languages as C++, Java, and Ada95. These languages are well-suited for framework implementation for several reasons. They offer a dedicated construct to represent abstract interfaces that play a crucial role in framework design because they define the interface between the architectural skeleton provided by the framework, and the customization components that are used to form instantiated application.

The presented RTJ Framework is implemented in Java. Java is in many respects better suited as an implementation vehicle for framework technology when compared to Ada or C++. Java and Ada do not use the pointer arithmetic that is present in C++. Furthermore, both Java and Ada do not have the complicated mechanism of multiple inheritance, but provide rather simpler semantics. Both have multi-threading mechanism integrated in the language and therefore can provide object synchronization, whereas C++ leaves multi-threading to the operating system. However, Java is superior to Ada because it leads to higher development productivity, it is more portable and its wide user base leads to lower development costs. Moreover, Java includes facilities for automatically generating documentation from the code. This feature was used extensively in the RTJ Framework. Although past versions of Java do not support the development of real-time applications, recently real-time versions of Java have been released.

3.2 Motivation

Java has the potential to become the language of choice for control software applications. The primary objective of the RTJ Framework implementation to a real-time version of Java is to evaluate maturity and suitability of real-time Java for the framework based applications from the control system domain.

3.3 Standard Java and Real-Time Java

The Java language was introduced by Sun Microsystems as a simpler and more robust version of C++. Its core principle is universal portability, the Java language was designed to enforce the “write-once-run-anywhere” paradigm (WORA). This is achieved through the introduction of a Java Virtual Machine (JVM) as a standard layer between the application code and the underlying hardware and operating system platform. However, ensuring compatibility with all of them inevitably resulted in weak specification at Java language level. It is obvious that standard Java is not suitable for embedded control applications both because it is poorly adapted to resource-constrained environments and because it does not support real-time programming. Although, most of the basic syntactical constructs that one normally associates with real-time programming like tasking, synchronization and event handling constructs are present, their semantics is not strong enough to build programs with the level of determinism and timing predictability that is essential in the real-time domain. Moreover, real-time applications need
a close interaction with the hardware, which is not always the case for desktop applications. In order to preserve WORA, Java applications are supposed to interact only with the JVM and the language does not offer facilities to directly manipulate memory, hardware registers or interrupt vectors. As a consequence, interrupt handlers and device drivers – basic elements of most real-time applications – are simply impossible to develop in Java. Finally, the presence of the garbage collector and the dynamic nature of the language make static analysis difficult or impossible.

These constraints resulted in a language that, when seen from the point of view of a real-time programmer, looks under-specified. However, extensions of the language are being explored that make it suitable for real-time applications too.

In Table 3.1 are summarized the main shortcomings of standard Java for real-time applications that have been identified during the implementation of the RTJ Framework.

Table 3.1 indicates that most of the inadequacies of Java for hard real-time stem from the weakness of its specifications, which is a consequence of the desire to keep the language compatible with as many operational environments as possible. The way to make Java compliant with real-time constraints is to develop a Java Virtual Machine that would guarantee predictable behavior. For example, to improve thread scheduling mechanism. The standard Java language does not provide any guarantee about how threads of different priorities are scheduled. On the other hand, the language does not forbid any particular scheduling policy. A real-time JVM therefore should provide an implementation that is suitable for real-time applications. This approach can be applied for all other shortcomings described in Table 3.1. Such a JVM would be compliant with the language specification and would provide real-time features.

3.3.1 Approaches to Real-Time Java

There are two mainstream solutions to the real-time Java problem (see Figure 3.1). The first one is based on individual solution of particular suppliers. The second one is provided by a group of suppliers that followed a formal definition of an extension of the Java language. The first milestone on this road was laid in 1999. The National Institute of Standards and Technology (NIST) prepared a report on the Requirements for Real-Time Extension for the Java Platform [18]. This document became the basis for the two efforts to produce a real-time Java specification.

Two parallel and competing efforts are under the way to provide a specification for a real-time extension of Java. One is carried out by Sun’s Java Community Process [83]. Its output is so-called The Real-Time Specifications for Java (RTSJ) [9] which were approved as a formal extension to the Java language in November 2001. The RTSJ covers all the shortcomings of standard Java identified in Table 3.1.

The second set of specifications is being elaborated by the J Consortium that brings together a group industry stakeholders that are independent of Sun Microsystems. The J Consortium specifications have not yet been finalized and appear to have less support within the user community. Their aim is a more compact run-time system with lower demands on hardware recourses [53]. There is an indication that the real-time Java extensions should be compliant with Ravenscar
### 3.3. Standard Java and Real-Time Java

<table>
<thead>
<tr>
<th>Issue</th>
<th>Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threading Model</td>
<td>Under Specified: no guarantees about dispatching policy of ready threads</td>
</tr>
<tr>
<td>Intra-Thread Synchronization</td>
<td>Missing Feature: no safe means to suspend a thread until a well-defined time in the future</td>
</tr>
<tr>
<td>Inter-Thread Synchronization</td>
<td>Under Specified: wait and notify facilities are present but order of release of notified threads is not specified</td>
</tr>
<tr>
<td>Access to Shared Resources</td>
<td>Under Specified: no priority inversion policy is specified for synchronized statement</td>
</tr>
<tr>
<td>Memory Allocation</td>
<td>Under Specified: memory is allocated from the heap but allocation policy is not specified and could be non-predictable and non-deterministic</td>
</tr>
<tr>
<td>Garbage Collection</td>
<td>Under Specified: GC collection algorithm and its triggering conditions are not specified and could be non-predictable and non-deterministic</td>
</tr>
<tr>
<td>HW Interrupt Handling</td>
<td>Missing Feature: no facilities for linking threads or event handlers to HW interrupts</td>
</tr>
<tr>
<td>Asynchronous Event Handling</td>
<td>Under Specified: standard Java event model adequate if threading model were adequate</td>
</tr>
<tr>
<td>Dynamic Class Loading</td>
<td>Incompatibility: dynamic class loading disrupts timing behavior</td>
</tr>
<tr>
<td>Class Initialization</td>
<td>Under Specified: no guarantee about the when classes are initialized</td>
</tr>
</tbody>
</table>

Table 3.1: *A list of shortcomings of standard Java for real-time applications.*

Currently there are several JVM software solutions compliant with RTSJ. Namely *JTime* by...
TimeSys that released the first JVM fulfilling RTSJ [84], AERO-VM developed by ESA and a consortium lead by Astrium SaS, Aicas and the University of Linkoping [40], OVM developed by a consortium of Purdue University, SUNY Oswego, University of Maryland, and DLTech [65], further jRate by Angelo Corsaro [23, 24], and FLEX by MIT [68].

Two real-time Java platforms, Jbed by Esmertec and PERC by NewMonics have been selected to perform component and system level testing of the RTJ Framework (see Section 3.4.6), they have been also used for a deployment of the instantiated applications (see Chapter 7). The two following sections discuss Jbed by Esmertec and PERC by NewMonics in more detail.

3.3.1.1 Jbed

Jbed is an embedded real-time operating system based on the Java programming language. The Jbed platform offers a nearly complete support of the JDK 1.2 APIs that guarantees the real-time behavior. Jbed schedules threads using a priority-based scheduling policy with a
priority inheritance mechanism to avoid unbounded priority inversion although more robust ceiling priority protocol [17] is not provided. In addition to standard threads, Jbed provides a custom class Task to encapsulate threads with more flexible scheduling and task synchronization policies. In the latest version 2.0, Task supports Earliest Deadline First scheduling (EDF) [17], but also allows the definition of periodic tasks. Class Task is integrated in the Java thread model because it is inherited from Thread class. Jbed provides an incremental automatic garbage collector that is scheduled as a low-priority non real-time background task. If a real-time task makes a request for memory that cannot be satisfied, the garbage collector is activated with the priority of the blocked task. Additionally to that, Jbed provides a very good framework for developing device drivers in Java and for interfacing to the hardware. In particular, it is possible to link release of a task to reception of a specific interrupt.

Experience with the Jbed platform in the RTJ Framework has been mixed, two major problems have been identified. Firstly, the RTJ Framework adheres to a policy of never using dynamic memory allocation in its real-time part. This policy has to be followed to avoid dependencies on a garbage collector, because its behavior is implementation specific and mostly not deterministic. Unfortunately, the Jbed documentation does not document usage of dynamic memory allocation in the Java libraries and run-time services. Strict adherence to the policy of no dynamic memory allocation is therefore impossible. Secondly, the RTJ Framework was basically developed as a cyclical system and most of its functionalities are usually implemented as periodic tasks. Additionally, there is a requirement that sensors and actuators should be sampled at specific times within a cycle, and hence the periodic tasks that control the sampling must be exactly phased. Jbed supports periodic tasks, but does not have a sleep_until facility. This makes it impossible to control accurately the phase of tasks.

Jbed execution model is illustrated in Figure 3.2. The Java class files are compiled to bytecode, and by means of the target byte-code compiler (TBCC) compiled and linked together with the Jbed VM and libraries into a single boot file for a particular hardware platform. The Jbed environment offers loading and reloading classes into a running Jbed system. This special feature can be used in situation when the system cannot be shut down, but not in real-time applications where the timing predictability plays an important role. This technique is also known as dynamic loading or hot loading.

The Jbed is no longer being developed or sold, and there are no similar products being developed by Esmentec.

3.3.1.2 PERC

PERC runs on top of a conventional RTOS. The PERC's runtime platform consists of the embedded real-time Java Virtual Machine called PERC Virtual Machine (PVM), Java libraries that are compliant with a subset of the JDK 1.3 APIs, and QuickPERC just-in-time (JIT) compiler. PERC provides round robin scheduling with priority inheritance for tasks with the same priority. PERC has a preemptable garbage collector running as a low priority background task that can be optionally switched off. PERC also consists of a set of development tools like PERCAccelerator, ROMizer, and PERCH.

PERCAccelerator is a tool that optimizes Java applications for the PERC virtual machine
(PVM). It produces a single accelerated class file that can be executed on different platforms. ROMizer is a tool that loads classes prior to runtime, dynamic loading of classes is not allowed. This has a positive effect on complexity of the PERC system, and memory requirements. There is also an improvement in execution speed, because programs run from ROM.

PERCH is a tool like javah, it produces a C/C++ header (.h) file, and a skeleton C or C++ source (.c or .cpp) file. The header and source files provide the skeleton for implementing any native methods declared in the class file.

PERC supports static and dynamic linking of native code. The dynamic linking is realized by System.loadLibrary("<library>"). The static linking requires ROMizer tool that preprocesses the classes containing the native methods.

PERC supports a large set of operating systems: Windows NT/2000, VxWorks, VxWork-sAE, VxSim, Linux, OSE, OSE Softkernel, and two processor architectures: x86 and PPC. Therefore an application can be debugged and tested on a desktop environment first and then ported to an embedded system.

Figure 3.3 shows PERC execution model. The Java class files are ROMized to an image file and linked with the JVM. Before generating the final image, the ROMizer compiles the class files into native code. There are two possibilities to link the native methods, one is to link it statically into the executable module, or to link them dynamically.

### 3.4 Implementation Aspects of the RTJ Framework

This section describes the general principles that guided and constrained the implementation process of the RTJ Framework.
3.4. Implementation Aspects of the RTJ Framework

### 3.4.1 Framework Components as JavaBeans

The RTJ Framework is a component-based framework. The term "component" designates an object that exposes one or more interfaces, through which communication with other classes takes place. The RTJ Framework is organized as a set of such components. The component standard provided by the Java technology is called JavaBean. The RTJ framework components are compliant with the JavaBeans standard, i.e., the framework is organized as a set of JavaBeans components. There are three practical consequences of the decision to implement the framework components according to the JavaBeans standard:

1. All classes have a default constructor, which is without parameters.
2. All component attributes are treated as JavaBeans properties, they are accessible only through setter and getter methods (see Section 2.6.1). The proper setup of properties is verified by the method isConfigured.
3. The RTJ Framework event model was designed to be compatible with the event model defined by the JavaBeans standard (see Section 2.6.1).

### 3.4.2 Real-Time Issues

There are at present two specifications for real-time Java. The first is proposed by the Real-Time for Java Expert Group and the second by the J Consortium. Since there is still some uncertainty about the maturity of these specifications, the RTJ Framework is designed to be independent of any real-time specific aspects of Java. To make the RTJ Framework compatible with different variants of real-time Java imposes the following restrictions on the implementation of the RTJ Framework:

- Use of the new operator is only allowed in constructors and in initialization methods, which are intended to be called only during the initialization phase of the application.
• No use of exceptions.
• The RTJ Framework components are treated as potentially shared and their non-initialization methods are synchronized.
• Only the java.lang and java.math packages are imported by framework classes. All other Java classes are excluded as potentially unsafe (for instance, it is not possible to guarantee whether they perform dynamic memory allocation or not).

3.4.3 Scheduling and Synchronization

The RTJ Framework components are designed to be used in a multi-threading environment. The components can be divided into two categories with respect to scheduling and synchronization issues:

**Active Components** – are the components that can have their own execution thread.

**Passive Components** – are the components that do not have their own execution thread. They are executed by active components or by other passive components.

The active components defined by the RTJ Framework are written to be independent of how often they are activated and of the order in which they are activated. Basically to each active component, a dedicated thread is associated. How the threads are scheduled is an instantiation issue.

Operation in a multithreading environment requires the implementation of mechanisms to preserve data integrity. In the case of the RTJ Framework this is done at three levels. At the first level, it is necessary to ensure that objects are accessed in mutual exclusion. For example:

```java
a.method();
```

An execution of the method `method()` might change the internal state of object `a`, therefore this method call can only be guaranteed to be thread-safe if its execution is not interrupted by other threads that perform operations on object `a`. This is achieved by marking all methods that may be called during the operational phase `synchronized`.

At the second level, it is necessary to ensure that data that are acquired from the same objects through separate operations are consistent. For example:

```java
v1 = a.method1(...);
v2 = a.method2(...);
```

Suppose that there is a requirement that the two return values `v1` and `v2` are mutually consistent in the sense that the state of object is not changed when they are acquired. The solution applied in the RTJ Framework is as follows:
synchronized (a) {
    v1 = a.method1(...);
    v2 = a.method2(...);
}

The acquisition operations are enclosed in a synchronized block. This ensures that the state of the a object, from which the data are acquired, is not changed by other threads.

There are two cases in the RTJ Framework where this situation arises. First, processing of framework events, which are stored in repositories. The event repositories are implemented as FIFO queues where the arrival of a new event overwrites the oldest event in the queue. There is no guarantee that for instance two successive operations performed on the same event reference will result in a consistent event repository content. Second, processing of object lists. The RTJ Framework provides classes that represent lists of objects. These classes offer iterator methods that allow retrieving all the objects in the list. The retrieval is typically done in a for loop. Inconsistencies may arise if, for instance a first thread performs the for loop, and a second thread inserts or removes items from the list.

At the third level, it is necessary to ensure that data acquired from several sources are consistent. A typical example is the processing of sensor data by the controller. There could be a situation where a controller processes data from two different sensors and produces data for two different actuators. For example, the sensor acquisition task is interleaved with the control processing task and the latter is interleaved with the actuator commanding task. It could happen that sensor data have been acquired at different times, or the actuators receive control action data that have been computed in two different cycles. This problem can be solved by a definition of a scheduling policy that ensures the proper execution order. In the example mentioned above the deadline of the sensor task has to be scheduled before the release time of the controller task, and the release time of the actuator actions after the deadline time of the controller task.

3.4.4 Framework Memory Model

Java offers a special mechanism for a memory management known as automatic garbage collection (GC). Standard Java provides GC that is not compliant with real-time behavior. Although, there are some real-time Java implementations provide a real-time garbage collector [1, 40, 84], the RTJ Framework does not use the garbage collector service at all. All memory required for the application based on the RTJ Framework is allocated during initialization and never released during the execution. To achieve this requires avoidance of dynamic memory allocation, which is heavily used in standard Java libraries, and in the standard Java event model (see the modified Java event model used in the RTJ Framework described in Section 2.5.2.2).
3.4.5 Integration of Non-Standard Components

In the context of the RTJ Framework are non-standard components understood as software units implemented in a language other than Java. For example, the RTJ Framework provides Matlab Wrapper that is an abstract Java class that acts as a general container for C procedures generated by Matlab Real-Time Workshop. The Matlab Wrapper relies on Java Native Interface (JNI), a technology that allows using C/C++ and assembly code in an application written in Java, and on the other hand ensures that the code is completely portable across all platforms.

3.4.5.1 Java Native Interface

The Java Native Interface is the native programming interface for the Java platform. The JNI allows use libraries written in languages other than Java, such as C, C++. Already the first release of the Java Development Kit (JDK) version 1.0 included the Native Method Interface (NMI) that allowed calling functions written in C/C++. The JNI appeared for the first time in JDK version 1.1. The current version of the JDK supports both the old Native Method Interface (NMI originally implemented in JDK 1.0), and extends JNI to incorporate new features in the Java platform. JNI is a part of the Java platform, which means that the application with the incorporated native code will run on all implementations of the Java platform [78].

The implementation of a Java class using native methods is a process consisting of several consecutive steps.

1. Design and implement a class that will use native methods. Such a class must contain declarations for native methods, and also include a static initializer that loads a library with the native code.
2. Compile the class declaring the native methods.
3. Generate header files by using the javah program. The header files provide a function signature for the implementation of the native methods.
4. Implement the native methods in C or C++. The implemented function must have the same function signature as the signatures generated by javah program.
5. Compile the header and implementation files into a shared library file (e.g. .dll for Windows, .so for Solaris).
6. Execute the Java application.

All these steps were applied for the development of several framework components encapsulating C code generated by Real-Time Workshop from Matlab/Simulink schemas modelling control algorithms.
3.4.6 Testing

Thanks to the "write-once-run-anywhere" paradigm (WORA) it was possible to develop and test the RTJ Framework and instantiated applications both on the desktop and on embedded processor. This process was performed at several levels and in several stages.

- Testing at a component level in the desktop environment
- Testing at a system level in the desktop environment
- Testing at a component level in the embedded environment
- Testing at a system level in the embedded environment

3.4.6.1 Testing at a Component Level in the Desktop Environment

Testing at a component level in the desktop environment levels aims at verifying the correctness of individual components. Tests are organized into several self-contained and independent test cases. A test case is encapsulated in a class derived from the **TestCase** abstract class (see the UML diagram in Figure 3.4).

![UML diagram of component testing](image)

**Figure 3.4:** The component testing – a customized Strategy design pattern.
Since most of the framework test cases require the same set up actions to be performed before the test execution begins, a class `TestCaseGenericSetUp` is provided to perform this common initialization actions and to serve as a convenient base class from which to derive specific test case classes.

The RTJ Framework provides a large number of default test cases that test various aspects of key components. The component testing mechanism is designed to be extensible, it means that a framework user can easily add new test cases for their custom components. The test case classes are called `TestCaseXxx` where `Xxx` is the name of the component to be tested. Sequences of test cases can be loaded into a `TestSuite`. A test suite allows several test cases to be executed in sequence as a single unit. `RegressionTest` executes all the test cases in a sequence.

The following sample code shows a code fragment of the `TestCase` class.

```java
public abstract class TestCase {

    public abstract class TestCase {
        public void setTestResult(boolean outcome, String failureMessage) {...}
    }

    public abstract void runTestCase();

    public boolean shutDownTestCase(){...}

    public boolean setUpTestCase(){...}
}
```

```java
public abstract class TestCase {

    public abstract class TestCase {
        public void setTestResult(boolean outcome, String failureMessage) {...}
    }

    public abstract void runTestCase();

    public boolean shutDownTestCase(){...}

    public boolean setUpTestCase(){...}

    public abstract void runTestCase();

    /**
     * It is called after the test execution. Returns false if * the shut down operation failed.
     */
    public boolean shutDownTestCase(){...}

    /**
     * It is called if the test starts. Returns false if * initialization of a test case fails.
     */
    public boolean setUpTestCase(){...}

    /**
     * It is called to execute the particular test.
     */
    public abstract void runTestCase();

    /**
     * It is called before the test starts. Returns false if * initialization of a test case fails.
     */
    public boolean setUpTestCase(){...}
}
```

Chapter 3. Implementation
3.4. Implementation Aspects of the RTJ Framework

3.4.6.2 Testing at System Level in the Desktop Environment

Two applications were instantiated to test the RTJ Framework functionality. The first application was developed to control the swing mass model (SMM), the second to control the ER1 robot by Evolution Robotics (see Chapter 7).

Before the prototype applications were ported on an embedded processor, they were tested on the desktop environment. Since the desktop environment differs from an embedded system in many aspects, there was a need to replace hardware dependent framework components by components that fit to a particular hardware configuration:

- The unit components (classes `AnalogSensor` and `AnalogActuator`) are replaced with two test unit components (classes `TestAnalogSensor` and `TestAnalogActuator`). The test sensor simulates a sensor that acquires a fixed value (which is hardcoded in the sensor class code). The test actuator sensor writes its output data to the standard output.
- The clock component (class `JbedSmmClock`) is replaced with a test clock component (class `TestSmmClock`) that retrieves time information using Java's standard `System.currentTimeMillis()` service.

Instantiation of the test applications uses the same instantiation factories as the real applications with the following exceptions:

- The factory `JbedSmmSystemFactory` is replaced with `TestSmmSystemFactory` to generate the test version of the clock.
- The factory `JbedSmmUnitFactory` is replaced with `TestSmmUnitFactory` to generate the test versions of the units (sensors and actuators).

The prototype instantiator component `JbedSmmInstantiator` is replaced with an instantiator component `TestSmmInstantiator` that is identical to the former except that it uses the test versions of the system and unit factories.

Since telemetry and telecommand data in the prototype application are sent and received through TCP/IP, the same code works in the desktop environment and therefore there is no need to replace the telecommand and telemetry components. Furthermore, the same ground station simulator can be used both for the prototype and for the test applications.

The scheduling policy adopted for the test application is the same as for the prototype application. Since the run-time system in a desktop environment has different characteristics from the Jbed or PERC run-time system, the application behavior can not be exactly reproduced. The test applications were used primarily to verify the functional correctness of the application code.

3.4.6.3 Testing at a Component Level in the Embedded Environment

The same set of test cases defined for the component-level testing in the desktop environment is executed in the embedded environment. The result must be the same as for the desktop environment.
3.4.6.4 Testing at a System Level in the Embedded Environment

The prototype application is loaded into the embedded system and executed. Two versions of the prototype applications were generated based on two different real-time Java implementations (Jbed by Esmertec and PERC by NewMonics).

3.4.7 Documentation Approach

The RTJ Framework is documented by comments in accordance with the JavaDoc conventions that are inserted in source files. Additionally, package overview documents are also provided and integrated in the JavaDoc documentation. The architectural-level description is generated automatically by the JavaDoc tool as a set of HTML pages describing the packages, classes, inner classes, interfaces, constructors, methods, and properties.

JavaDoc is a tool for generating API documentation. The JavaDoc output is customizable by JavaDoc doclets. Standard doclets generate HTML format API documentation, custom doclets can generate any type of output format, such as SGML, XML, RTF, PDF, or PS [78].

3.5 Conclusions

The first conclusion is that the Java language is a natural implementation vehicle for software frameworks and is attractive because of its safety features and ability to make the framework code compatible with multi-threaded operations. It provides the JavaBeans standard that is supported by several environments and that facilitates framework instantiation (see Section 5.2). On the other hand, standard Java is at present unsuitable for real-time systems primarily because it does not support real-time programming.

The second conclusion is based on a survey of suppliers of real-time Java implementations. It has been presented that real-time compliant versions of Java capable of supporting the needs of real-time applications are available as commercial products and that there are already versions compliant with formal extensions of the Java language to cover real-time needs. Two real-time Java platforms, Jbed by Esmertec and PERC by NewMonics were successfully used for purposes of the RTJ Framework testing. Several new Java Virtual Maschine implementations have appeared on the market recently: JTime [84], Aero JVM [40], OVM [65], jRate [23, 24], and FLEX [68]. All are compliant with The Real-Time Specification for Java produced by The Real-Time for Java Expert Group [83]. Presumably they provide a better solution than Jbed and PERC, such as a real-time garbage collector, a full control over real-time and normal threads, and direct access to any memory areas.

Third, the main advantage of using Java for control system programming is that the presence of the Java Virtual Machine (JVM) erases the distinction between the desktop and the embedded environment. Typically control software development is done on embedded targets that are not user friendly. However, JVM separates the application from the underlying platform and from the OS. This allows almost all the development to be done on a convenient desktop environment. The experience with RTJ Framework certainly confirms that Java's
“write-once-run-anywhere” claim holds for control systems as well.
CHAPTER 4

Introduction to Instantiation Problem

Framework instantiation is the creation of a real instance or particular realization of an abstraction represented by a software framework. The instantiation process is a tedious and error-prone task. In this chapter we focus on facilitation of the whole application development process.

Contents

4.1 Introduction ....................................................... 66
4.2 Motivation ....................................................... 66
4.3 Instantiation Process ........................................... 66
4.4 Instantiation Approaches ................................. 67
  4.4.1 Naive Instantiation Approach ................. 67
  4.4.2 Factory Instantiation Approach .............. 67
  4.4.3 Automated Instantiation Approach .......... 71
  4.4.4 Feature-Based Instantiation Approach ...... 72
4.5 Conclusions .................................................... 72
4.1 Introduction

The framework instantiation is a process transforming a software framework into a concrete application. It takes place in two steps, firstly, the application-specific components required by the application are constructed with guidance and constraints from the design patterns. Secondly, the framework components and the application specific components are instantiated, configured and composed together to construct the final application. The instantiation approaches described in this chapter covers only the second step. The first step was already covered in two previous chapters dedicated to framework design and implementation (see Chapter 2 and 3).

4.2 Motivation

The instantiation process for the RTJ Framework consists of a long sequence of instructions that instantiates, configures the framework and application specific components, and composes them together. Coding and testing this sequence is a conceptually simple but physically tedious and error-prone task. The main objective for the development of different framework instantiation concepts was to facilitate or even automate this process.

The RTJ Framework covers the domain of control systems. Developers or engineers working in this domain are used to design their systems in environments like Matlab or LabView that provide user friendly GUI-based tools to define the control algorithms, to model and simulate their behavior and finally to generate their implementation for example as optimized C code. Although, such tools excel at modelling of control algorithms they are not designed to cope with system management tasks such as the unit management, failure detection or failure recovery, etc. In fact, there is no commercial tool that provides sophisticated abstractions to model all aspects of a complex control system. Hence, the second motivation was to introduce a concept and a tool with a capability to model framework based applications in graphical, user friendly environment.

For adopting new instantiation concepts in the practical usage is important to provide low cost solution. Therefore, the third motivation is use only mainstream technology and tools.

4.3 Instantiation Process

The instantiation process of any software framework is the proper transformation of a given application specification into an ordered set of instantiation operations, the execution of which will result in the creation of an application that implements the initial specifications.

In the case of the RTJ Framework, the instantiation of an application from the framework consists in performing an ordered sequence of the following six instantiation operations:

- instantiation of a framework component
- setting the value of a component property
4.4 Instantiation Approaches

- setting the value of an indexed property
- setting the value of a static property
- linking an event-firing framework component to an event-listening framework component
- adding a component to an object list

In fact, the definition of an application instantiated from the RTJ Framework is reduced to the definition of an ordered set of operations of the type listed above. All the instantiation operations can be expressed in terms of the methods declared by the external interfaces of the framework components. The instantiation sequence can therefore be encoded as an ordered set of method calls performed upon the framework components. It is important to stress that the order in which the instantiation operations are performed is important. Not all sequences of instantiation operations result in meaningful or legal target applications. The semantics of the RTJ Framework requires instantiation of some components before others, and it requires some types of operations to be performed according to certain rules.

4.4 Instantiation Approaches

This section provides an overview of four framework instantiation approaches that were used for realization of applications instantiated from the RTJ Framework, however they are easily applicable to other software frameworks as well. Two of them, Automated and Feature-Based Instantiation Approach described in Section 5.2 and 6.3 are innovative methods that significantly facilitate the framework instantiation process.

4.4.1 Naïve Instantiation Approach

The instantiation process where the application developer instantiates one component after the other and then makes their configuration and composition manually, and has to solve the order in which the instantiation operations are performed might work for small frameworks (up to 50 components) but not for bigger frameworks like the RTJ Framework (255 default components). Therefore, this so called naive approach which is widely used needs to be replaced by more sophisticated approaches that would minimize manual work, and coding mistakes. One such approach that allows the developer to concentrate on bigger parts of the instantiated application rather than each component individually is introduced in the following section.

4.4.2 Factory Instantiation Approach

The RTJ Framework provides a set of framework factories to perform the creation and configuration of framework core components, and ordering of instantiation operations. The construction and configuration of application specific components is be done in dedicated factories
that are provided by the application developer and that extend the framework factories. Additionally, the framework provides a configuration class that defines all the constants required for the configuration of the framework core components. The application developer should define a similar configuration file to define the application specific configuration constants.

The actual configuration process is implemented by an application specific Instantiator class. The concept of the Instantiator class and factories is based on the abstract factory design pattern [47]. As an example see the class diagram of the simplified version in Figure 4.1 where two application specific instantiate classes implement makeControllerFactory and makeTelecommandFactory methods declared in the abstract Instantiator class. They create application specific factories derived from the abstract ControllerFactory, TelecommandFactory classes that are further responsible for creation of application specific components.

![Diagram showing the instantiation process](image)

Figure 4.1: The instantiate concept.
4.4. Instantiation Approaches

4.4.2.1 Instantiation Factories

An instantiation factory is a component that is responsible for creating and configuring a set of components. For each framework functional aspect there exists one framework instantiation factory provided by the RTJ Framework and one application specific factory provided by the developer.

Two types of instantiation factories are required for the RTJ Framework instantiation process:

**Framework Factories** – create and configure the core components of the RTJ Framework. Therefore, they are application independent and are provided by the RTJ Framework itself. The framework factories are realized as abstract classes, which provide implementation of methods to create and configure the framework core components. In addition, they may declare abstract methods to create application specific components. The declaration of these methods can be seen as a support for the development of the application specific factories made by the developer.

**Application Specific Factories** – create and configure the application specific components of an application. Therefore, they are provided by the application developer. Typically they extend the framework factories, but can be developed independently as well. The application factories which extend the framework factories have to provide implementations for the abstract methods declared by the framework factories. They can also provide additional methods to construct components whose type could not be expected at the framework level.

The development of the factories for the RTJ Framework stick to the following principles:

- The methods that create components are called make<component name>
- Factories are independent of each other.
- Factories should create and configure components.
- Factory methods should return components with generic framework types.

The instantiation factory design is based on the abstract factory design pattern [47]. The RTJ Framework defines 11 framework factories, particularly ControllerFactory, Failure-DetectionFactory, FailureRecoveryFactory, ManoeuvreFactory, ModeFactory, MonitoringFactory, SystemFactory, TelecommandFactory, TelemetryFactory, and UnitFactory. Classes that implement the framework factories are stored in the Factories package.

The concept of the framework and application factories is demonstrated in Figure 4.2, it focuses on instantiation of application specific control components. Figure 4.3 shows an instantiation sequence diagram the emphasizes the time ordering of control and telecommand factory method calls.
Chapter 4. Introduction to Instantiation Problem

Figure 4.2: The instantiation factories.

Figure 4.3: The sequence diagram of the instantiation process.
4.4. Instantiation Approaches

4.4.2.2 Configuration Variables

Configuration of framework and application components by concrete values is a specific instantiation operation that is a part of the whole instantiation process. The concrete configuration values can be directly set to each instantiated component (e.g. the maximum number of listeners for each event source, the number of operational modes for each mode-dependent component, etc.). However, the better solution is to concentrate all configuration values into specific classes, because some of these configuration variables must be defined for all applications instantiated from the framework.

The RTJ Framework provides a configuration class called FrameworkConfiguration that defines the configuration variables of the framework core components that can be edited by developers when they instantiate a particular application. Application developers should additionally define a similar class to store application specific configuration variables.

4.4.2.3 Application Specific Instantiator

The Instantiator class where the configuration process should be implemented is not provided by the RTJ Framework, because it is application specific. An application developer should implement an application specific Instantiator class that performs following actions:

- It instantiates the concrete framework factories.
- It calls the creation and configuration methods defined by the factories.
- It finalizes the configuration process in cases when components could not be completely configured within a factory.

4.4.2.4 Case Study

The factory instantiation approach was tested on the RTJ Framework. The objective of the test was to develop different control systems instantiated from the RTJ Framework for two laboratory models, a simple mechanical system known as the swing mass model (SMM) and the ER1 robot system. The applications are described in detail in Chapter 7 dealing with test cases.

4.4.3 Automated Instantiation Approach

The automated instantiation approach is a new concept based on generative programming techniques [27], which demonstrate automating the instantiation process of a component-based framework. The process is automated in the sense that designers configure and assemble the framework components using intuitive visual operations in a GUI-based environment. Their configuration actions are then used to automatically generate the framework instantiation code with respect to ordering constraints.
Generative techniques for framework instantiation are not new but tend to rely on domain-specific languages or on bespoke specification encoding and compilation techniques. Though effective and powerful, they are comparatively complex and present a high barrier to entry for general users. The distinctive feature of the approach, described in Chapter 5, is instead its simplicity and its reliance on mainstream technology and tools.

4.4.4 Feature-Based Instantiation Approach

The objective of the feature-based instantiation approach based on feature modelling technique is as in the case of automated instantiation approach to develop a generative environment for software framework instantiation. The environment must be generic in the sense of being able to support different frameworks. It means that it must be built upon a framework meta-model rather than upon a particular framework. This avoids the cost of having to develop a dedicated generative environment for each target framework. The environment must consequently be configurable with a model of the target framework. Such a model is required for two purposes. First, to parameterize the generative environment with the framework to be instantiated. Second, to serve as a basis for specifying the application to be instantiated. The framework model must therefore contain all the information needed to configure the framework assets to instantiate a target application and it must allow the target application to be specified. The full description of the proposed instantiation approach can be found in Chapter 6.

4.5 Conclusions

This chapter introduces the framework instantiation problem and provides an overview of framework instantiation approaches that were used for realization of applications instantiated from the RTJ Framework.

The factory instantiation approach represents one “automated” way to handle the instantiation problem. The use of design patterns offers a view on the framework instantiation problem that is more abstract and at higher level. The approach isolates a developer from implementation classes. The developer manipulates only factory classes that are responsible for implementation of component instantiation processes. The benefit of this approach is significantly faster implementation of the target application with reduced code errors in the instantiation part. The other advantage regards to exchanging application functionality. Since each factory is present in the instantiation code only once, it makes it easy to replace the concrete factory for a new one.

Two new approaches automating the instantiation process of software frameworks, the automated and feature-based instantiation approach, were mentioned. They differ in the way the instantiation information is encoded, in how this information is processed, and in the scope of the instantiation process. Further information discussing the new approaches in details can be found in Chapter 5 and 6.
This chapter presents two automated instantiation environments based on generative techniques. The central part of the automated instantiation environment is a graphical user interface that allows users intuitively to configure and assemble framework components, a simulator that validates the constructed application, and a code generator that automatically generate the instantiation code.

Contents

5.1 Introduction .................................................. 74
5.2 Automated Instantiation Environment .................. 74
  5.2.1 Instantiation Environment .......................... 77
  5.2.2 Required Technologies ............................... 86
5.3 Generic Instantiation Environment ....................... 87
  5.3.1 Framework Description Model ...................... 88
  5.3.2 Implementation Aspects ............................. 88
5.4 Case Study ................................................... 89
5.5 Conclusions .................................................. 91
5.1 Introduction

The automated instantiation approach is based on generative programming techniques [27]. To develop a generative programming environment for applications within a certain domain is necessary to complete 5 steps:

1. Domain scoping.
2. Definition of a common architecture for applications in the domain.
3. Development of configurable and customizable components to support implementation of the domain architecture.
5. Development of a code generator to automatically transform an application configuration description into source code instantiating and configuring the pre-defined components.

Steps 1 to 3 were, for the most part, covered in two previous chapters (see Chapter 2 and 3). Step 2 is equivalent to the development of a software framework for a certain domain. In step 3 the focus is on the implementation of domain functionalities, and also the integration of code coming from other sources. Steps 4 and 5 are introducing new concepts of the framework instantiation process based on generative programming techniques that result in an user friendly instantiation environment.

The straightforward way to implement the five steps mentioned in the previous paragraph is to define a domain-specific language whereby the application specification can be expressed, and then to build a compiler that allows these specifications to be translated into source code. However, this solution does not satisfy the requirements of the low cost solution using the mainstream technology and tools. The approach that was finally selected is based on a special environment where the developer expresses the requirements graphically with the support of context specific information provided by the environment. The environment is then responsible for translating the requirements expressed by the developer into an instantiation sequence. The advantage of this approach is that the developer concentrates only on the modelling/development of the target application, while the environment takes over the derivation process of the application.

5.2 Automated Instantiation Environment

The automated instantiation approach based on generative programming techniques is shown in Figure 5.1. The primary inputs to the automated instantiation process are the framework components. The target application is developed using graphical components hosted in the “component composition environment” where they are configured and composed together with other components. However, the framework components are not usually compatible with the component model required by the graphical environment and therefore cannot be easily imported in the environment. Several steps have to be done before the developer can manipulate
with the framework component in the user friendly graphical environment.

Firstly, the component properties and methods that participate during the configuration process are encoded to a formal description using XML. In the case of the RTJ Framework it represents the six types of instantiation operations described in Section 4.3. The RTJ Framework components have form of Java classes. For each Java class, an XML document is generated that describes the instantiation relevant information of the component that the class represent. These XML documents are called visual proxy descriptor files.

The visual proxy descriptor files are automatically constructed by a dedicated Java application. Naming conventions (see Section 2.6.1) are used to define the behavior of the class during the instantiation phase. The application that constructs the visual proxy descriptor files therefore only needs to have access to the API implemented by a class. In fact, the application is a parser that processes all the methods implemented by the class, and by checking whether their signature conforms to certain patterns, determines which properties it exposes, which events it fires or processes, and which object lists it manages. This information is then encoded as an XML document.

Visual proxy descriptor files need to be constructed for the framework concrete classes because
these are the only ones from which framework components can be instantiated. In addition, proxy descriptor files must also be constructed for abstract classes and abstract interfaces. Figure 5.2 illustrates the component encoding process.

![Formal Component Description Diagram](image)

Figure 5.2: The component encoding process.

The application designer interacts with the graphical instantiation environment during the application development process. The environment allows him to configure the framework components and to assemble them to construct the target application. It would be desirable to provide the composition environment that directly manipulates the framework components. However, experience showed that this is not practically feasible, because of several reasons: the target components may not be visualizable, they may be written in languages other than Java, their instantiation operations may not fit the JavaBeans component model, etc. Therefore, an alternative approach was taken where the framework components were replaced with proxy components that emulate their behavior during the instantiation phase. These proxy components are called visual proxies, because they visually represent the real framework components. The visual proxies are the components that are imported in the composition environment and are the components that are actually manipulated by the designer. They should expose the same properties, listen to the same events, fire the same events, and expose the same adder methods as its original application component. The visual proxy components are constructed from their visual proxy descriptors, using XSL programs. Since the composition environment is based on a JavaBeans bean builder tool, the visual proxies implements JavaBeans component model. In order to facilitate their manipulation, beaninfo components, bean editor components, and a bean customizer are provided. The bean customizer is common to all visual proxy beans whereas the bean editors and the bean-info are specific to each visual proxy bean. They are also generated by a XSL program, see
5.2. Automated Instantiation Environment

Figure 5.3. The visual proxy components are then ready to be imported into the composition environment.

Figure 5.3: The transformation of visual proxy description to JavaBeans component model.

5.2.1 Instantiation Environment

The instantiation environment consists of three parts: a component composition environment where the designer configures and assembles the components to construct the target application; a simulator that validates the constructed application by an execution of operations upon the configured application components; and a code generator that converts configuration information of the target application to the instantiation code.

5.2.1.1 Component Composition Environment

The component composition environment is a graphical environment where the developer uses a drag & drop mechanism to place components on a composition sheet, configures and assembles them to construct the target application. As explained above, the designer manipulates the visual proxies of the framework components. These visual proxies are implemented in Java and have the form of visualizable JavaBeans components. For this reason the natural choice for the composition environment is to select a standard bean builder tool. Several bean builder tools were evaluated, however, those supplied with commercial Java packages (e.g. Visual Age, JBuilder, VisualCafe, or CodeWarrior) are often complex and
cannot be customized by the user. The best suited tool allowing customizable configuration was the **Bean Builder** from Sun [80].

The component composition environment is the customized version of Sun’s **Bean Builder**, see Figure 5.4 presenting a screenshot of the composition environment. The top window offers palettes with the framework components. The bottom right window is the composition form where components are displayed and visually manipulated. The connection lines represent composition relationships between components. The component configuration is done using the property sheet in the bottom left window. In addition to that, special wizards are provided (not shown in the figure) to handle non-standard configuration actions (e.g. the additions of items into object lists). The wizards and the property editors are also enforcing the constraints on the instantiation process (e.g. constraints on the range of values of certain variables).

The main objective followed in Sun’s **Bean Builder** customization process was to minimize as far as possible code changes on the original **Bean Builder** classes, because of compatibility with future releases of the **Bean Builder** or with releases of bean builder tools provided by other vendors. There are several ways to change the bean builder behavior without changing its code:

- The bean builder allows the user to load a custom root container, which is the component
that represents the composition sheet where the instantiated components are dropped by
the developer.

- The bean builder uses an external XML file to define the components that are available
to the developer. This file can be easily edited to add new components or to change the
way they are organized.

- The property setting process can be customized by using property editors. The bean
builder offers a small set of default property editors but additional ones can be loaded
through beaninfo components, which have no impact on the bean builder code.

- The component configuration process can be configured by associating bean customizer
components to the framework components. The bean customizers are loaded through the
beaninfo components and therefore have no impact on the bean builder code.

Apart from a small number of minor changes that are implemented to improve the user-
friendliness of the component composition environment, the suggestions mentioned above
cover all requirements on the compatibility with its future releases.

5.2.1.2 Code Generator

The operations performed by the developer in the component composition environment (plac¬
ing components on a composition sheet, configuring and assembling them) result in the defi-
nition of an application configuration. The code generation problem can be seen as the trans-
formation problem of the application configuration into source code. In the proposed approach
this is done in two steps.

First, when all instantiation and configuration actions are completed then the target applica-
tion is encoded in a XML document by means of the long-term persistence mechanism [79].
The long-term persistence mechanism is a feature of Java version 1.4 that allows saving of a
component state as a list of component configuration operations. By default, Java provides
XMLEncoder class that performs long-term persistence encoding mechanism. However, the
default encoder does not cover requirements for applications instantiated from the RTJ Frame-
work resulting in need to develop customized encoder. The customized encoder expresses the
application configuration as a set of instantiated framework components whereas the devel-
oper manipulated with the visual proxy components. Furthermore, the order in which the
configuration operations are performed upon the framework components must satisfy certain
ordering constraints. The customized encoder is integrated into the component composition
environment, which makes it easier to use and also satisfies the idea of automated instantiation
environment.

Second, the simplest way to implement the code generation is to use an XSL program to process
the encoded application configuration XML document and produce an instantiation sequence.
The output of this XSL program is a single Java class called Instantiator. This class
provides a method called configure() that implements all the configuration actions de-
finied by the developer as Java code in the component composition environment. An execution
Chapter 5. Automated Instantiation

of this method causes instantiation and configuration of the application components. The code generation process is shown in Figure 5.5. Figure 5.6 presents an example of automatically generated instantiation code that is precisely the same as code that would be obtained by the so-called naive approach (see Section 4.4.1).

5.2.1.3 Simulation

Simulation plays a crucial role in commercial tools like Matlab. The developer builds a system by configuring predefined blocks and then by means of other tools automatically generates the code that implements the defined design. One of the strengths of Matlab—like environments is that they let the developers test the application they are building by simulating it. Simulation and design are interleaved and developer can, at every step in the design process, pause and perform simulations to test selected aspects of the applications he is building.

It might seem that the simulation facility is not relevant to software frameworks, however, experience shows the opposite. For example, the developer may want to test whether all instantiated components are properly configured, or to perform a selective execution of operations on some of the components within the instantiation environment and to monitoring their observable states. Simulation of software frameworks or more precisely instantiated applications can be seen as a special debugging tool that helps to developer verify whether their configuration actions satisfy their requirements. The goal is to allow simulation of an incompletely configured
application, because debugging is especially valuable during the configuration process. One of the benefits of the graphical component composition environment for software frameworks based on Sun's Bean Builder is an integrated design mode and run mode. The design mode is a default mode used for component instantiation and configuration. The run mode causes an execution of the instantiated application. However, the default run mode is not very useful, because it executes components placed on the composition sheet. These components are visual proxy components that accept configuration operations but do not provide any other functionality, i.e. their execution has no effect. On the other hand visual proxy components correspond to framework components and each configuration action performed upon them has an equivalent in the framework components. In the previous section dedicated to code generation, the procedure of encoding the configuration actions in a XML document by the long-term persistence mechanism is described. The final code generation is performed by a XSL program writing the instantiation commands in a Java class called Instantiator. Except the generation of code to a file, the reflection mechanism is used. The same mechanism is applied for a dynamic construction and configuration of the target application components. When the run mode is selected, the transformation process is executed and framework components are dynamically instantiated and configured (see Figure 5.7). The developer can then select a simulation task (simulation modes are discussed in following paragraphs). The simulation request is redirected to the underlying framework components. The state of the framework components can then be inspected to examine the effect of the simulation action. The simulation concept offers three types of simulation modes: configuration check, repository check, and the state inspection simulation.

```
... 
sysMan = new RtjAocsFramework.SystemManager.SystemManager();
sysMan.setMaxNumberOfSystemListeners((int)2);
clock = new RtjAocsFramework.OperatingSystemObjects.DummyAocsClock();
staticObj = new Utilities.StaticBean();
staticObj.setMaxNumberOfFailureListeners((int)5);
staticObj.setMaxNumberOfConfigurationListeners((int)10);
staticObj.setAocsClock(clock);
sysMan.setMaxAocsObjects((int)100);
staticObj.setSystemManager(sysMan);
aocsObj = new RtjAocsFramework.BasicObjects.AocsObject();
nullRec = new RtjAocsFramework.FailureRecovery_NULLRecoveryAction();
resetRec = new RtjAocsFramework.FailureRecovery_SystemReset();
tmList = new RtjAocsFramework.ObjectList_TelemeterableList();
tmList.setListSize((int)2);
clock.setTelemetryFormat((byte)2);
staticObj.setDestructionRecoveryAction(nullRec);
sysMan.setMaxReconfigurable((int)1);
...
```

Figure 5.6: An example of automatically generated instantiation code.
The configuration check is the simulation facility that checks an internal configuration of the instantiated components and reports whether they are configured or not. Almost all RTJ Framework components implement the isConfigured() operation which actually performs the configuration check. This operation usually checks whether component properties are set and have legal values, all references point to some object, and other component specific configuration actions. The simulation by configuration check is very useful at any part of the development phase, but the most appropriate is the end of the instantiation process.

The component composition environment based on Sun's BeanBuilder integrated the configuration check in the composition sheet (root container). By switching to the run mode and pressing the "Check" button on the composition sheet the configuration check is executed. It
can be executed at any time during the configuration process. When it is executed, the environment dynamically instantiates the components that the designer has pulled down on the composition form and it dynamically configures them as their proxies on the composition form are configured. After the application has been created, method isConfigured() is called on all components. Depending on the result of the method call, the color of the icons representing the components on the composition form is changed. The following color code is used:

**Blue** – the configuration check has not yet been executed  
**Red** – the configuration check reports "the component is not configured"  
**Yellow** – an exception was thrown while performing the configuration check  
**Orange** – the instantiation of the component to be checked has failed (this is typically due to the class implementing the component not being accessible to the loader class)  
**Green** – the configuration check was performed successfully, the component is properly configured

The repository check is the second simulation facility. Its purpose is to visualize any configuration events that are generated during the application configuration process. Configuration events are generated by RTJ Framework components when illegal configuration actions are performed upon them. The presence of configuration events is an indicator of errors that occurred during the application configuration process. It is therefore recommended to inspect the event repository where configuration events are stored and to verify that no events were generated during the configuration process.

The implementation of a repository check is again based on the idea of dynamically instantiating the components that the developer has pulled down on the composition form. When the application is completed, the state of the configuration event repository is inspected and results are displayed in a dedicated window (the components that created the configuration events are listed together with the event identifiers). The repository check is always automatically performed whenever a configuration check is performed.

![Figure 5.8: An evaluation of the repository check simulation.](image)
The third simulation facility called *state inspection* is more general than the previous two. This type of simulation allows a selective execution of methods on some of the components within the instantiation environment and the monitoring of their states. A concept of the general simulator presented here is structured in the following parts:

- A selection of methods that should be executed, and properties that should be inspected.
- Byte code modification of methods that should be executed, and properties that should be inspected. The byte code modification lies only in modification of accessibility modifiers.
- Encoding the target application in a XML document, and a dynamical construction and configuration of the target application from framework components modified by the byte code modifier.
- A generation and initialization of the simulator GUI.
- Linking the simulator GUI with the target application.

First, the user selects certain methods that he wants to execute and certain properties that he wishes to inspect (see Figure 5.9). Before the environment creates a simulator GUI tailored to perform these methods and to inspect those properties, it modifies the target components to make the necessary state information accessible by means of a byte code modifier [22]. In practice, the *private* and *protected* accessibility modifiers are turned into *public*. It is then up to the user to use the simulator GUI to manually perform the methods in any desired sequence. The user can specify a number of simulation steps, an activation order, and whether the selected method is executed only once or cyclically (see Figure 5.10).

The general simulator can be started directly from the composition sheet (root container) by pressing a button with the label *Simulate* if the component composition environment is in the run mode.

### 5.2.1.4 Instantiation Support

The component composition environment provides a mechanism of integrating the framework-specific information that is instantly available during the instantiation and configuration phase in order to assist the developer with instantiation problems. The current release of the component composition environment offers a simplified version of the concept that provides API information of framework components displayed as HTML pages.

The proposed instantiation assistance approach is based on two main principles:

1. The information for the instantiation help must avoid all duplication of raw information.
2. The existing sources of information must be exploited as much as possible.

The following information type and format should be provided by the environment:

- The general information about the component composition environment.
Figure 5.9: Selection of components and their methods and properties that should be simulated.

- The information about individual components and about their properties and information about abstract interfaces, etc. This can be achieved when the code is commented by the JavaDoc tags. However, given the special needs of the environment, a customized doclet should be used. For example, a design pattern tag to associate a component to the design patterns whose instantiation it supports, or an instantiation operation tag to mark operations that denote instantiation operations that may come into play during the
framework instantiation process, etc. [21, 88, 91].

- The most appropriate way to display the information is as HTML pages that can be viewed through a standard browser.

### 5.2.2 Required Technologies

The automated instantiation environment is based on three key technologies. Namely, **XML** for encoding information to a format that is easily treatable, **XSL programs** for automatic code generation, and **JavaBeans** standard for integrating components into the graphical component composition environment. Since these technologies are well known and supported, one of the initial requirements was satisfied.
5.3 Generic Instantiation Environment

The generic instantiation environment is based on ideas of automated instantiation environment introduced in Section 5.2. The objective of the generic environment is to allow a developer to instantiate a software framework or components of any type graphically. It means to select an appropriate component from component libraries, to place the component on a workspace area, to configure its properties and to connect the component to other components. The generic environment introduces a framework description model that replaces the concept of JavaBeans which are used as a means for manipulating relevant instantiation information of framework components in the automated instantiation environment and presents a new graphical instantiation environment (GIE) [49]. The overall approach of the generic environment is shown in Figure 5.11.

The RTJ Framework was used to validate the concept of the generic instantiation environment. The RTJ Framework components are represented by Java classes. For each Java class, an XML document is generated that describes the instantiation relevant information of the component that the class represent. The XML documents are further transformed to the framework description model required by the generic environment. The framework description model is

![Figure 5.11: The generic instantiation approach.](image)
defined as a set of Java interfaces that must be implemented for each framework. A direct transformation from framework components implemented by any object-oriented language to framework description model is possible as well, and probably in most cases preferable (in fact this approach is sketched in Figure 5.11). The transformation of the RTJ Framework components to XML documents and then to a framework description model is a technique inherited from the previous concept. On the other hand, it demonstrates a versatility of the previous concept, because the formal description of framework components in the XML format can now be very easily transformed by a XSL program to the new description model. The GIE then allows to the developer to visually construct the target application. The configuration actions performed upon the instantiated components forming the target application are encoded and stored in a XML document representing a formal configuration description, which is used for generation of the instantiation code. The format of the XML document containing the formal configuration description is expressed through XML Schema. Since the instantiation and configuration actions are stored in a XML document, the simplest and cheapest technology that generates instantiation and configuration sequence is again XSL – a language for transforming XML documents.

5.3.1 Framework Description Model

The framework description model required by the GIE (illustrated in Figure 5.12 as a feature diagram) is defined as a set of Java interfaces (see Figure 5.13). It consists of a component and connection description model and configuration knowledge. The component description model contains information about the component type, component properties that can be initialized by default values and constrained to a feasible range of values, and the visual representation in the environment. The connection description model contains information about the connection type, i.e. a description of what components can be connected with this particular connection, and it contains information about the visual representation of the connection in the environment as well. In addition to that, a configuration knowledge can be included to the framework description model. The purpose of the configuration knowledge is to provide simple instantiation support. For example, to navigate which component can be connected to some other component.

The design of the framework description model was influenced by a unified component metamodel described in [71].

5.3.2 Implementation Aspects

The generic Graphical Instantiation Environment (GIE) [49] is developed as a plug-in for the Eclipse platform [33], see Figure 5.14. It is managed by the Graphical Editing Framework (GEF) [34] – a framework for the development of graphical editors. The GIE requires a compliant framework description model, which implements the framework model description interfaces (see Figure 5.13). The development of the framework description model for a particular framework is facilitated by a set of abstract classes known as adapters that provide default implementation of some operations. Therefore, a framework description model can be easily
implemented as just an extension of the adapter classes. The preview of the GIE graphical user interface is shown in Figure 5.15.

5.4 Case Study

The automated instantiation environment described in Section 5.2.1 is a fully functioning instantiation environment for the RTJ Framework. In order to demonstrate its effectiveness, the control application for a simple swing mass laboratory model (SMM) was developed. SMM consists of two rotating disks connected with a torsional spring. A DC motor drives one of the disks. The goal of the control application is to control the speed and position of the other disk. The detailed information about the control application can be found in Chapter 7.

The generic instantiation environment described in Section 5.3 was used for the development of a control application for the ER1 robot [73]. The objective of the control application is to drive the ER1 to a desired position defined in Cartesian coordinates. The application is described in detail in Chapter 7.
Figure 5.13: The framework description model [49].

Figure 5.14: The graphical instantiation environment architecture.
5.5 Conclusions

A generative approach to framework instantiation was proposed and demonstrated. The distinctive feature of the introduced approach is its simplicity and reliance only on mainstream technologies. Two graphical instantiation environments facilitating the development process of target applications and several parsers and XSL code generators were developed.

A downside of the approach is a certain lack of generality: the instantiation environment described in Section 5.2.1 and all parsers and code generators are to some extend targeted at one particular framework. On the other hand, the generic instantiation environment described in Section 5.3 provides a component composition concept cannot offer the simulation facility that is a very powerful feature during the development phase, because it can test selected aspects of the applications. However, the proposed generative approach provides several guidelines that
would facilitate its porting to other frameworks. The justification for this solution is a belief that, given the wide variety of frameworks and the lack of standardization at the framework level, there is more practical value in providing a blueprint for the development of a simple, though framework-specific, instantiation environment than in constructing a complex general purpose, instantiation environment for a generic framework.

Simplicity and reliance on mainstream technologies are important features that promise to bring the automated instantiation approach within the reach of most framework designers and users. For example consider the development time, here is provided an estimate of how much effort would be required to port the approach to another framework. The total development time was approximately 10 man-months, included the test case study. Since this is a new concept, much of this effort went into false starts and trying out new ideas. The re-implementation of the concept for a new framework which complies with the proposed guidelines requires at most 4 man-months. Of course, the estimate is heavily dependent on the way the framework is designed. If a generative approach to the instantiation process is presumed, it is essential that this issue should be kept in mind already during the framework development phase.
CHAPTER 6

Feature-Based Modelling and Instantiation

This chapter describes a feature modelling technique aimed at modelling and instantiation of software applications built from software frameworks. The presented feature-based modelling concept improves traditional feature modelling approaches in several respects, and in addition to that it also provides instantiation facilities.

Contents

6.1 Introduction .............................................. 95
6.2 Feature-Based Framework Modelling .................. 95
   6.2.1 Feature Model .................................. 95
   6.2.2 Modelling Approach ............................ 95
   6.2.3 Framework Meta-Model ......................... 98
   6.2.4 Application XSD Generator ................... 100
   6.2.5 Constraint Model ............................... 101
   6.2.6 Summary ....................................... 102
6.3 Feature-Based Framework Instantiation ............ 103
   6.3.1 From Application Model to Instantiation Code .. 103
6.1 Introduction

The feature modelling concept presented in this chapter distinguishes from the approach described in [5, 27, 54, 55] in several respects. It defines a feature meta-model that avoids the need for normalization found in other meta-models, for example [5, 27, 86]. It also offers a means to decompose a large feature diagram into extensible and self-contained modules. It defines an XML-based approach to express the feature models that offers a low-cost path to the development of a support tool for building the models. It proposes an XSL-based mechanism to express complex composition rules for the features. Finally, it supports both the modelling of the software framework and of the applications instantiated from it.

6.2 Feature-Based Framework Modelling

In general, feature modelling is a creative activity of modelling the common and the variable properties (features) and their dependencies [27, 76]. The output of the feature modelling process that is performed during the domain analysis phase is a feature model.

6.2.1 Feature Model

A feature model [5, 25, 27, 48, 54, 55] is a description of the relevant characteristics of some entity of interest. The most common representation of feature models is through FODA – style feature diagrams [27, 54, 55]. A feature diagram is a tree-like structure where each node represents a feature, and each feature may be described by a set of sub-features represented as child nodes. Various conventions have been introduced to distinguish between mandatory features (features that must appear in all instantiated applications) and optional features (features that are present only in some instances). Limited facilities are also available to express simple constraints on the legal combinations of features.

Figure 6.1 shows an example of a feature diagram for a system representing (much simplified) control systems. The diagram states that all control systems have a single processor, which is characterized by its internal memory size, and have one to four sensors and one or more actuators. Sensors and actuators may have a self-test facility (optional feature). Sensors are either speed or position sensors whereas actuators can only be position actuators.

6.2.2 Modelling Approach

Traditional feature modelling approaches are usually based on a two-layer structure with a meta-modelling level, which defines the types of features that can be used and mutual relationships among them, and a modelling level where the feature model for the entities of interest is constructed. However, in the context of a generic environment where an application is automatically instantiated and configured from a framework, three levels of modelling are needed:
Chapter 6. Feature-Based Modelling and Instantiation

Figure 6.1: A feature model example.

- Framework Meta-Modelling Level
- Framework Modelling Level
- Application Modelling Level

The framework meta-modelling level describes a domain of software frameworks. The framework meta-model is fixed and independent of any particular software framework. At the framework modelling level, a particular software framework is described. A framework must be an instance of the framework meta-model. Finally, at application modelling level a particular application is described as an instance of the framework. The application model represents a specification of the application to be instantiated from the framework.

With respect to the available software technologies the XML language is used to express the feature models and XML Schema to express the meta-models. The relationship of instantiation between a model and its meta-model is expressed by saying that the XML-based model must be validated by the XML Schema that represents its meta-model. The resulting modelling architecture is shown in Figure 6.2.

6.2.2.1 Feature Composition Rules

The framework feature model is represented by features and their relationship introduced in the framework modelling level, and by rules expressing legal combinations of features that can appear in an application model.

However, the current feature modelling techniques are often weak in the definition and checking of combination rules. They are usually well-equipped to express local composition constraints, namely constraints on the combinations of sub-features that are children of the same feature. Thus, for instance, it is easy to express a constraint that a certain feature can only
6.2. Feature-Based Framework Modelling

![Diagram](Image)

**Figure 6.2: A XML-based feature modelling approach.**

have one sub-feature or that it can only have one sub-feature selected out of two possible options. It is instead harder to express *global composition constraints*, namely constraints based on relationships between features in different parts of the feature diagram. The FODA notation contained require-exclude relationships (expressing the condition that the presence of a certain feature is incompatible with, or requires, the presence of another feature in a different part of the feature diagram). With the exception of the *pure::variants* approach [6, 69], however, more complex kinds of global composition constraints are not covered.

The use of an XML-language to express the application model opened a new way to more sophisticated approaches expressing global composition constraints. They are expressed through a feature diagram of the same kind that is used to model a software framework. The provided compiler translates the constraint model expressed as a feature diagram into an XSL program that checks compliance with the constraints at the application model level. This is illustrated in Figure 6.3. The concept of global composition constraints is discussed in detail in Section 6.2.5.

### 6.2.2.2 Feature Macros

*Feature macros* offers a mechanism to split a large feature diagram into smaller modules that can be used independently of each other. Feature macros resemble the macro facilities provided by some programming languages to encapsulate segments of code that can then be “rolled out” at several places in the same program. A feature macro represents a part of a feature diagram consisting of a node together with all its sub-nodes. The framework meta-model allows a feature macro to be used wherever a feature can be used. A large feature diagram can thus be constructed as a set of independently developed modules. Note that the same feature macro can be used at different points in a feature diagram (see Figure 6.4 for an example). Feature macros thus provide both modularity and reuse. The feature macro mechanism is similar to the module concept of [5] but goes beyond it because, in order to enhance reuse potential,
Inheritance-like extension mechanism has been added. Given a feature macro \( B \) (for “Base”), a second feature macro \( D \) (for “Derived”) can be defined that extends \( B \). Feature macro \( D \) is an extension of feature macro \( B \) in the sense that it adds new features to the feature sub-tree defined by \( B \). This allows a form of reuse where a feature sub-tree can be parameterized and instantiated for use in different parts of the same feature diagram with different requirements. This is more flexible than just allowing a feature sub-tree to be used in the same form at different places in a feature diagram. The analogy with inheritance is only partial because the possibility of overriding existing sub-features in a base feature sub-tree is not provided: new sub-features can be added but existing ones cannot be deleted or modified.

### 6.2.3 Framework Meta-Model

The framework meta-model is constructed as follows (see Figure 6.5). There are two main building blocks, it is a feature and a macro. A feature can have sub-features but the connection between a feature and its sub-features is through the use of a group. The group gathers together a set of features that are children features of some other feature and that are subject to local composition constraints. The same feature can have several groups attached to it.

#### 6.2.3.1 Feature and Group Cardinality

Both features and groups have cardinalities. The feature cardinality defines the number of instances of the feature that can appear in an application. The group cardinality defines the number of features chosen from within the group that can be instantiated in an application. Cardinalities can be expressed either as fixed values or as ranges of values. The distinction between group cardinality (the number of distinct features within the group that can be instan-
6.2. Feature-Based Framework Modelling

Figure 6.4: A feature macro example.

Figure 6.5: The framework meta-model.
tiated in an application) and feature cardinality (the number of times the feature can be instantiated in an application) reduces the multiplicity of representation and the consequent need for normalization found in the feature meta-models proposed by other authors [5, 27, 58, 86]. Figure 6.6 shows an example of framework model. The feature “ControlSystem” has three groups: “Sensors”, “Actuators”, and “Processors”. The group “Processors” has cardinality 1 and has only one sub-feature with default cardinality 1. This means that the sub-feature is mandatory (it must be present in all applications instantiated from the framework). Since the feature cardinality is 1, then only one instance of the feature may appear. The group “STypes” has two sub-features and cardinality 1. This means that its sub-features are mutually exclusive. The cardinality of the “Sensor” feature is a range cardinality (1, 4) which implies that the feature can be present in an application with up to four instances.

Figure 6.6: A framework model example.

6.2.4 Application XSD Generator

The application XSD generator is an XSL program that processes a framework model to generate a meta-model for the applications instantiated from that framework (see Figure 6.2). Conceptually, an application model can be seen as a feature model where all variability has been removed, namely as a feature model where all features are mandatory and where all features have cardinality of 1. In this sense, the feature meta-model generated by the application XSD generator is simpler than the meta-model of Figure 6.5 because it does not include the group
mechanism and the feature cardinality mechanisms. In another sense, however, it is more complex. The framework model specifies the types of features that can appear in the applications and the local composition constraints to which they are subjected. The application XSD generator must express these constraints as an XML Schema using the XSD language. Basically, this is done by mapping each feature group in the framework meta-model to an XSD group and by constructing an XSD element for each legal combination of features in the group. This implies a combinatorial expansion and an exponential increase in the size of the application meta-models. It is, however, noteworthy that the computational time for applying the XML Schema only need increase linearly with the number of features in the framework model. The practical experience with application meta-models derived from framework models containing about a hundred features is that the computational time for enforcing the meta-model remains negligible and compliance with the meta-model can consequently be checked in real-time and continuously as the user selects new application features within a standard XML tool.

![Instance of the framework model](image)

**Figure 6.7: Instance of the framework model.**

### 6.2.5 Constraint Model

Figure 6.8 shows the constraint model that was defined in order to express global composition constraints. This model is an instance of the framework meta-model of Figure 6.5. It covers three types of global constraints. The first two are the traditional “requires” and “excludes” constraints [54, 76] which are covered by the “Requires” and “Excludes” features. The former states that the feature “Feature” requires the “Required Features” features to be present. The latter states that the features “Feature” are mutually exclusive. The “Custom Condition” feature
models the third type of global constraint. This allows very general XPath expressions to be used to express any generic constraint on the combination of features and their values. The “Element” feature represents either a toggle feature or a valued feature. Each “Element” has a “Name” (by which it is referenced in “Condition”) and a “Value” that uniquely identifies the feature (or its value) using XPath syntax. The “Condition” feature is expressed as a logical expression of the above elements or as an arithmetic expression where some operands are the values of the above elements.

Figure 6.8: *The constraint model.*

### 6.2.6 Summary
A software framework is characterized by two types of models:

**Framework Model** – describes the mandatory and optional features of applications within the framework domain together with their local composition constraints (see Figure 6.2).

**Constraint Model** – describes the global composition constraints on the framework features (see Figure 6.3 and Figure 6.8).

The described feature modelling approach was used to develop a feature model for the whole RTJ Framework. The RTJ Framework feature model is encoded in XML document that is validated by the framework meta-model expressed in XML Schema. A part of the RTJ Framework feature model is demonstrated on the Unit Management Architecture described in Section 2.7.1.2.
6.3 Feature-Based Framework Instantiation

This section presents the framework instantiation approach based on the feature modelling technique for software frameworks introduced in Section 6.2. The objective of this instantiation approach is to perform application instantiation from the application feature model, which is derived from the framework feature model. The output of such an instantiation process is a description of the application components in a format that is suitable for direct import to a component composition environment where the components can be further easily configured. For example consider the component composition environments described in Section 5.2.1 and 5.3.

6.3.1 From Application Model to Instantiation Code

The key idea is to use the framework model that is defined during the domain analysis phase in the framework creation process as a basis for constructing an application model in the framework instantiation phase. This framework model describes all possible variants allowed by the framework. Application specification during the framework instantiation phase can be done by selecting the features in the framework that are desired to be included in the target application. This pruning of the framework model results in the construction of an application model that can be seen as a specification of the target application. The transition from framework to application model is done automatically by generating an application meta-model from the framework model. The application meta-model is entirely framework-specific and defines the framework within which applications can be specified. The approach is sketched in Figure 6.9. The application model is derived by instantiating a meta-model that is automatically generated from the framework model. The presence of the application meta-model enforces the consistency of the application model. The framework and application models are encoded in XML documents, and the application meta-model is implemented as an XML Schema. Therefore, it was natural to implement a transformation of the application model to corresponding instantiation code as an XSL program. In fact, the XSL program is assembled from several easily replaceable XSL modules that cover different instantiation aspects. The goal of this instantiation process is not to generate complete code, but to generate a description of the application components in a format that is possible to import in a component composition environment where the instantiated components covering application requirements are further configured. Although the configuration actions can be easily done in the component composition environment, some of them are performed during the transformation process driven by the XSL program. The XSL program is also responsible for construction and implementation of components that are not provided as default framework components, they implement framework interfaces or abstract classes and therefore their default implementation can be provided.

6.3.2 Instantiation Information for Application Code Generation

The software assets offered by the software framework are seen as a set of building blocks that can be used to construct applications from the framework, see Figure 6.10.
Two kinds of building blocks are considered: components and abstract interfaces. Building blocks have a name and are typed. Component building blocks additionally may have attributes that define their characteristics, and these can be of different types. The simplest kind of attributes are properties (the term is used in the JavaBeans sense) but attributes can also represent operations that a building block may implement or they can represent aspect, i.e. cross cutting concerns that are shared by several types of components. The functionality attribute is provided to represent more abstract attributes that cannot be reduced to a property, operation, or aspect.

The objective of the type information shown in Figure 6.10 to the meta-model shown in Figure 6.5 is to perform a mapping of the framework features to specific software items provided by the framework as building blocks to construct target applications. Obviously, some features may be too high-level to be thus mappable. For this reason, the type mechanism of Figure 6.10 introduces the composite type. The type information is added to the domain model during the domain design and the domain implementation phases. The goal is to provide an extended framework model where each terminal feature can be implemented by some specific software item provided by the framework.

Since the instantiation of an application will often require the development of application-specific components to complement the default components predefined by the framework, the type mechanism of Figure 6.10 assigns a status to each component. Components may be provided (the component is predefined by the framework), generable (the component is not prede-
6.3. Feature-Based Framework Instantiation

fine by the framework but the framework offers a generator that can automatically generate it), or missing (the component must be developed from scratch). This frees application designers from the need of restricting themselves to components that are offered by the framework. One important side effect of this approach is that the definition of the type information during the domain design phase forces the framework designer to check that all framework features are covered by the framework assets.

As indicated in Figure 6.9, the complete application model with the full type information is used to generate the application code, in our case a description of the instantiated application components in a format that can be imported in a component composition environment. The order in which the code generation process is executed is following.

1. The instantiator XSL program performs instantiation and partial configuration of features that have a component type with a provided status.
2. Features that have a component type with a generable status are transformed to components with generated implementation.
3. Features that have a component type and a missing status are transformed to stub components. Their implementations have to be completed manually, but the stubs can be used in an early prototype phase. At the end of this phase, all the components required by the
target application are present.

4. The instantiator XSL program runs features of aspect type and apply aspect transformations to adapt the application components to the specific needs of the application.

### 6.4 Tool Support

At the time of writing there is no graphical tool which fully supports the feature modelling concept described in Section 6.2 and framework instantiation from the application feature model described in Section 6.3. Thus the concept and test cases were implemented using the XmlSpy [3] and oXygen [81] tools, which are simple, but sufficient XML editors. This became a motivation for development a new graphical feature modelling tool compliant with the introduced concepts. The tool is currently under the development, but a prototype version was already released [89]. The new graphical feature modelling tool is provided as an Eclipse [33] plug-in that requires Eclipse Modelling Framework (EMF) [32], and Graphical Editing Framework (GEF) [34]. The tool allows the user to build a feature diagram consisting of features, supports constraints, macros, and cardinalities, which are not supported by existing tools [35, 74, 69]. It also allows the user to prune the framework feature model in order to define an application feature model. A screenshot of the prototype version of the advanced feature modelling tool is shown in Figure 6.11.

![The graphical feature modelling tool.](image)
6.5 Case Study

The approach was tested on the RTJ Framework. The objective was to instantiate and pre-configure components that after importing into the component composition environment introduced in Section 5.2 were further configured towards to build a control application for the swing mass model (SMM). The application is described in detail in Chapter 7.

6.6 Conclusions

The feature-based framework modelling and instantiation concept has been tested on the RTJ Framework. The instantiation information is encoded by means of the building block mechanism in a feature model obtained as an extension of the framework model. This information is further processed by a XSL program that generates an instantiation code format that can be imported in the component composition environment, for example in the environment described in Section 5.2.1 and 5.3.

Feature modelling techniques are often used to describe variability within a family of software products, but the distinction between the specification model captured in the domain analysis phase, and the implementation model that describes the configuration variability of the assets created in the domain design and implementation phases is rarely done. Most automated instantiation approaches take an instantiation model as the input for the generation process, although [7] is a notable exception. The proposed approach tries to build a bridge between these two different but complementary views by deriving the instantiation model as an extension of the specification model.

Many approaches use dedicated utilities or languages e.g. XVCL [75] for managing variants in component-based families [93] or FDL [86] to describe variability in software assets in [85].

Framework feature models are rather large systems, and its construction and maintenance would certainly have been more difficult without the feature macro mechanism (see Section 6.2.2.2). There were a few parts of the RTJ Framework feature diagram that were sufficiently similar to be treated as instances or extensions of the same feature macro. This introduced a new degree of reuse in the framework feature model that helped manage its complexity.

The proposed mechanism for expressing global composition constraints (see Section 6.2.2.1 and 6.2.5) allows description of the global constraints separately using the same notation and environment as is used for the definition of the application and framework models. This is significant advantage in comparison with other approaches where the global constraints are incorporated directly into the feature diagram.

A very important issue for the feature modelling process is the tool support. The definition of the framework and application models was done in an XML editor configured with the XML Schema that implements the respective meta-models. High quality tools, like oXygen [81] or XmlSpy [3], enforce compliance with the meta-model. For example, users are offered only choices of the legal features and their legal values at each step of the feature model editing
process. Compare to meta-modelling concepts provided by GME [52, 57] or EMF [14, 32] is the introduced XML based concept more powerful in expressing feature models and easier to use.

As in the case of automated instantiation concept introduced in Chapter 5 the distinctive feature of the feature-based framework modelling and instantiation concept is its simplicity and reliance on mainstream technologies. Here is an estimate of how much effort would be required to implement the feature-based framework modelling and instantiation concept for another framework. The total development time was about 8 man-months. The biggest effort went to the extension of the feature modelling concept (Section 6.2) and the description of the framework in terms of features. The complexity of this approach increases with a demand on the final code. The feature-based instantiation approach presented in Section 6.3 focuses on providing a description of application components in a format importable in a component composition environment. Much more effort would require a demand of the automatic generation of the complete instantiation code from the application feature model. An approximate estimation for the re-implementation of the feature-based framework modelling and instantiation concept for another framework is at most 3 man-months.
This chapter presents three control applications for two plants deployed on three platforms instantiated by the proposed instantiation approaches from the RTJ Framework.
7.6.2 The Control Application for the Swing Mass Model Deployed on the PERC Platform ........................................... 121
7.6.3 The Control Application for the ER1 Robot Deployed on the Java Platform ...................................................... 126

7.7 **Factory Instantiation Approach** ........................................ 129
7.7.1 The Control Application for the Swing Mass Model Deployed on the Jbed Platform ........................................... 129
7.7.2 The Control Application for the Swing Mass Model Deployed on the PERC Platform ........................................... 131
7.7.3 The Control Application for the ER1 Robot Deployed on the Java Platform ...................................................... 132

7.8 **Automated Instantiation Approach** .................................. 133
7.8.1 The Control Application for the Swing Mass Model Deployed on the Jbed Platform ........................................... 133
7.8.2 The Control Application for the ER1 Robot Deployed on the Java Platform ...................................................... 135

7.9 **Feature-Based Instantiation Approach** ............................ 137
7.9.1 The Control Application for the Swing Mass Model Deployed on the Jbed Platform ........................................... 138

7.10 **Conclusions** ............................................................. 139
7.1 Introduction

After the successful testing of the RTJ Framework's components it was a natural step to test the RTJ Framework on more complex applications (the testing concept is described in Section 3.4.6). As the framework is intended to cover a set of software applications from the control system domain, it is obvious to select target applications that fit this criterion.

7.2 Motivation

The development of control applications instantiated from the RTJ Framework for two different real-time Java platforms (Jbed, PERC) and for standard Java platform, for two different plants (SMM and ER1 robot) by different instantiation techniques (see Chapter 4) was a challenging task for several reasons:

- First, the need to build fully functional control applications from the RTJ Framework.
- Second, to prove the Java's WORA paradigm “write-once-run-anywhere” is also applicable for embedded real-time systems.
- Third, to test maturity of the Java language for real-time applications.
- Fourth, to prove the proposed framework instantiation concepts.
- Last but not least the instantiated application should serve as a tutorial for producing other framework based applications.

7.3 Overview of Test Cases

Figure 7.1 shows a matrix presenting an overview of realized control applications for different plants and platforms instantiated using different approaches. Rows represents control applications for particular plants (SMM or ER1 robot) deployed on one of three platforms (Jbed & PowerPC, PERC & VxWorks & PC, Java & PC). Columns represent the applied instantiation approaches (factory, feature based, and fully automated instantiation approach). A cross point of rows and columns denoted by a hook represent a realized control application developed for the particular plant and platform instantiated by the particular approach. Altogether six instantiation processes building control applications were performed. Although some control applications were instantiated more than once, they had always the same objectives and functionalities. Specifically, the control application for SMM deployed on Jbed & PowerPC was instantiated by all three approaches and the control application for ER1 robot deployed on Java & PC was instantiated by two approaches.
7.4 Controlled Plants

The RTJ Framework described in Chapter 2 and 3 and the instantiation concepts introduced in Chapter 4, Section 4.4.2, 5.2 and 6.3 were tested on two models. The first one is a laboratory model known as the swing mass model (SMM) developed by ETH Zürich, the second one is the ER1 robot developed by Evolution Robotics [41].

7.4.1 The Swing Mass Model

The SMM is a standard laboratory equipment for testing control algorithms with hardware-in-the-loop. It consists of two rotating disks connected with a torsional spring. The objective of the control action is to control the speed and angular position of the right-hand disk by acting on the left-hand disk. The system actuator is an electric motor which can drive the position and angular speed of the left-hand disk. The system sensors are position and angular speed sensors on the right-hand disk. A second motor is available that can apply a torque to the right-hand disk to simulate the presence of a disturbance torque. The SMM apparatus is shown in Figure 7.2.

7.4.2 The ER1 Robot

The ER1 Personal Robot System by Evolution Robotics is a low cost robot kit allowing to a developer to assemble a custom robot. The default configuration shows the robot carrying a laptop, shown in Figure 7.3 and 7.4. The kit already provides a software application to control all functions of the ER1 robot, including a vision and recognition system, acting based on a schedule, moving autonomously, playing sounds and music, or taking pictures and video. The robot’s complex behavior can be achieved by a combination of these individual behaviors.
Figure 7.2: The swing mass model (SMM).

Figure 7.3: The ERI robot.

Figure 7.4: The ERI robot carrying a laptop.
An interesting feature of the robot kit is provided a remote control mode. The remote mode enables to access the digital inputs and outputs and analog inputs of the robot using provided APIs, which can be used by most programming and scripting languages including C, C++, Perl, Java, Visual Basic. Unfortunately, there is no support for a direct access to low level hardware. Therefore the functionality of control applications is limited to the commands provided by robot’s APIs that are processed by the robot software first and then applied on the robot hardware. This has been found as a big disadvantage for the development of a control application based on the RTJ Framework whose behavior should be real-time compliant.

7.5 Platforms

7.5.1 Jbed & PowerPC

The Jbed RTOS based on the Java programming language running on a RPX Lite board mounting a PowerPC 823 microprocessor was used for a deployment of the instantiated applications based on the RTJ Framework. The Jbed is described in Section 3.3.1.1 in detail. The hardware set-up is shown in Figure 7.5 and 7.6.

Figure 7.5: The RPX Lite board with PowerPC 823.

Figure 7.6: The A/D and D/A converter board.
7.5. Platforms

Figure 7.5 shows a RPX Lite board settled by a PowerPC 823@66MHz microprocessor. Figure 7.6 is a board with an A/D and D/A converter. The converter board was used to interface the RPX Lite board to actuators and sensors of the swing mass model (SMM). The ranges of the analogue inputs and outputs are 0 – 12V.

7.5.2 PERC & VxWorks & PC

The other platform used for testing the framework instantiation concepts consists of a standard personal computer (PC) Intel Pentium III 1 GHz microprocessor with a Me2600i PCI A/D and D/A conversion board by Meilhaus Electronic GmbH, VxWorks real-time operating system by Wind River [90], and PERC Virtual Machine by NewMonics [60]. It should be stressed that the data storage system of the target PC (i.e. a hard disk drive) was not used, all data were directly loaded into memory. The PERC Virtual Machine version 3.3 described in Section 3.3.1.2 required VxWorks 5.4/Tornado 2.0.2.

The main advantage of the described configuration was the compatibility of PERC Virtual Machine with other operating systems, for example with Windows2000. This allowed testing and debugging applications on a user friendly desktop environment before porting to an embedded system with minor modifications (drivers).

Figure 7.7: The personal computer connected to the SMM through the use of PCI A/D and D/A conversion board.
7.5.3 Standard Java & PC

Java 2 Standard Edition (J2SE), Windows operating system and a standard personal computer (PC) Intel Pentium M 1.6 GHz and 512 MB of RAM were used for debugging and testing non-real-time aspects of the instantiated applications or for the applications that do not require real-time behavior.

The benefit of this configuration was a simple and fast deployment of the target applications, and more than sufficient support of development tools.

7.6 Instantiated Control Applications

7.6.1 The Control Application for the Swing Mass Model Deployed on the Jbed Platform

The goal of the control application for the swing mass model (see Figure 7.2) instantiated from the RTJ Framework for the Jbed & PowerPC platform described in Section 7.5.1 is to control the speed of the right-hand disk. The control application was instantiated by three different instantiation approaches: the factory instantiation approach described in Section 7.7.1, the automated instantiation approach described in Section 7.8.1, and the feature instantiation approach described in Section 7.9.1. The instantiated application includes:

- two operational modes
- failure detection checks on the main system variables
- autonomous failure recovery actions autonomously executed upon detection of failures
- provision of housekeeping data
- processing of external commands
- capability to execute speed profiles

It is therefore representative of a full control system as conceptualized in Figure 1.1. More detailed description about the application is provided in the following sections.

7.6.1.1 Description of the Control Application

The control application is described in terms of configuration actions performed on particular components in order to achieve the required behavior of the control application. The description can found in Section A.1, Appendix A.

7.6.1.2 Scheduling Approach

The prototype application uses the following framework active components:

- telecommand loader
• telecommand manager
• controller manager
• failure detection manager
• failure recovery manager
• manoeuvre manager
• unit manager
• telemetry manager

Except for the unit manager, all other active components are instantiated only once. The unit manager has two instances. The first instance, the sensor unit manager, is used to acquire the measurements from the application sensor. The second instance, the actuator unit manager, is used to send the actuation command to the application actuator. The control cycle for the prototype application has a fixed period with duration of 1 second. The only scheduling constraints that apply to the scheduling of the above active components are the following:

• the sensor unit manager must be triggered at the beginning of a cycle and must terminate execution within 50 ms from the beginning of the cycle (this is necessary to ensure that the sensor measurement is collected sufficiently early in the cycle)
• the actuator unit manager must be triggered not earlier than 800 ms in the cycle (this is necessary to ensure that the actuator is stimulated sufficiently late in the cycle).

In view of the above and in view of the capabilities of the Jbed run-time system, the scheduling approach selected for the prototype application is articulated by the following four real-time threads:

• Periodic Task 1
• Periodic Task 2
• Telemetry Thread
• Telecommand Thread

All four tasks have fixed priorities: priority 10 for the Periodic Tasks and priority 5 for the Telemetry and Telecommand Threads. The Jbed run-time system schedules threads using a fixed priority scheduler with pre-emption and priority inheritance. The priority level of 10 is the highest in the Jbed system and ensures that a thread is never preempted. The basic idea of the proposed scheduling scheme is that all operations that require timing determinism are performed in the (high priority) periodic threads whereas the (lower priority) telemetry and telecommand threads are used for operations that are not real-time safe. In particular, the management of the FTP socket that supports the telemetry/telecommand (TM/TC) link is performed in the telemetry and telecommand threads. In order to ensure that there is no disruption of the scheduling of the periodic threads, the interaction between them and the telecommand and telemetry threads is kept to a minimum and is tightly controlled to ensure
that potentially blocking calls that are performed from the telemetry/telecommand threads on objects that are used by the periodic threads have a predictable and limited blocking time. The *Periodic Task 1* thread defines the control cycle for the prototype application. Its main function is to trigger the following framework active components (in the order in which they are listed):

1. sensor unit manager
2. telecommand manager
3. controller manager
4. failure detection manager
5. failure recovery manager
6. manoeuvre manager
7. telemetry manager

After all the above functionality managers have completed execution, the periodic task goes to sleep for an interval computed to ensure that it is activated with a period of 1 second (this is the control period of the SMM application). The sleep request is implemented using the standard *Java sleep* service.

The *Periodic Task 2* thread runs with a period of 1 second but its release instant is 800 ms after the release instant of *Periodic Task 1* and is used is to trigger the actuator unit manager component. The periodicity of activation is achieved using the same mechanism based on the *Java sleep* service used for *Periodic Task 1*.

The *Telemetry Thread* runs with a lower priority than the periodic threads and its function is to send the telemetry buffer to the remote operator through the *FTP* socket. This is done by activating the telemetry sender active component (see the telemetry policy in Section A.1.6 of Appendix A). Ideally, the *Telemetry Thread* should be activated immediately after the *Periodic Task 1* has completed and should complete before *Periodic Task 2* begins. This would ensure an orderly forwarding of telemetry data: the telemetry manager (which is activated by *Periodic Task 1*) fills in a telemetry buffer, which is then emptied by the telemetry thread. This ideal case is illustrated in Figure 7.8.

![Figure 7.8: Jbed scheduling policy.](image-url)
However, the Telemetry Thread performs two operations that are not real-time safe:

- it forwards telemetry data over a data network that is shared with other users and where timeliness of service is not guaranteed
- it performs dynamic memory allocation operations because when a telemetry buffer is retrieved from the telemetry stream component for flushing, a new, empty, buffer must be created and supplied to the telemetry stream

Hence, the possibility exists that the telemetry stream will not finish its execution before the Periodic Task 2 is ready to start. In such a case, the Jbed scheduling mechanism ensures that the telemetry thread is preempted and that Periodic Task 2 is executed. The telemetry thread will resume execution in the next period. This may cause one telemetry buffer not be forwarded to the remote operator (loss of one frame of telemetry).

The synchronization between the Periodic Tasks and the Telemetry Thread is performed as follows. The Telemetry Thread goes to sleep on a call to wait after it has forwarded one telemetry frame. It is awaken by Periodic Task 1 that sends it a notify signal after executing all its functionality managers.

The Telecommand Thread runs with a lower priority than the Periodic Tasks and its function is to collect telecommands from the FTP socket. Whenever it is awoken, it checks whether a new telecommand has been received and, if so, it loads it into the telecommand loader (see the telecommand policy in Section A.1.5 of Appendix A). The telecommand loader will dynamically create an object to encapsulate the telecommand and will then load it into the telecommand manager. The telecommand receiver then goes to sleep for a period of 1 second. The telecommand receiver thread runs completely asynchronously from all other application threads. If the remote operator sends two telecommands in between two successive awakenings of the telecommand receiver thread, the earlier of the two telecommands will be lost. Processing of telecommands could be made more deterministic by linking its release to the end Periodic Task 1 and by raising its priority to the same level as the Periodic Task 1 and Periodic Task 2.

Note that the Jbed run-time system always creates an additional thread to run the garbage collector. This thread normally runs in the background with lowest priority. However, if a task makes a request for a memory allocation that cannot be serviced due to a lack of free memory, the garbage collector is invoked with a priority equal to that of the calling task. Finally, when Jbed is running under the control of a remote terminal (as would be the case for testing), two additional threads are created by the run-time system to manage the debugger connection to the remote terminal.

Finally, it must be stressed that the implementation proposed above relies on the standard Java sleep service to activate the periodic threads. This mechanism is known to be unsafe because of the possibility of preemption between the time when the calling thread computes the sleep interval and the time when it actually makes the call to method sleep. The ideal solution to this problem is to have a sleep_until method. This, however, is not provided by Jbed.

As an alternative solution, the two periodic threads could be run with priority 10 which is the highest priority foreseen by Jbed and that ensures that they are never preempted.
Note that Jbed provides primitive constructs to implement periodic threads (through the Task class that extends the Thread class). This mechanism was tried, but various implementation problems were encountered and eventually it was decided to revert to standard threads.

### 7.6.1.3 Memory and CPU Measurements

Memory and CPU usage measurements were performed on the instantiated control application for the SMM (see Section 7.4.1) deployed on the Jbed & PowerPC platform (see Section 7.5.1). The memory measurements are summarized in Table 7.1. The table compares the measurements for the control application instantiated from the RTJ Framework with the measurements for the simple “Hello World” program. This is a test program that simply writes “Hello World” to the standard output and is used as a reference to allow differentiation between user code and run-time code. The figures in Table 7.1 give total memory usage including both code and data.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>SMM control application</th>
<th>Hello World</th>
</tr>
</thead>
<tbody>
<tr>
<td>reflection and dynamical loading</td>
<td>1700 KB</td>
<td>1080 KB</td>
</tr>
<tr>
<td>no reflection and dynamical loading</td>
<td>938 KB</td>
<td>615 KB</td>
</tr>
<tr>
<td>no reflection and no dynamical loading</td>
<td>522 KB</td>
<td>194 KB</td>
</tr>
</tbody>
</table>

Table 7.1: Memory measurements of the SMM control application and “Hello World” program deployed on the Jbed & PowerPC platform.

In order to unriddle the figures in the table, one has to bear in mind the Java execution model implemented by Jbed (see Section 3.3.1.1). The Java class files are compiled to native code and linked with the JVM. Two basic linker options affect the memory requirements of an application. The noreflect option determines whether reflection information is included in the executable module. The data in the table shows that its inclusion can nearly double memory occupation. Its benefit is the possibility of using the Java meta-language facilities. The second option concerns the inclusion of the module to perform dynamic class loading. As shown by the table, these modules take about 400 KB.

The figures in the table indicate that the basic Jbed run-time system takes less than 200 KB. This figure excludes reflection and dynamic loading which would typically not be used in a real-time application. The application specific code for the SMM control application instead takes a little over 300 KB (excluding again reflection and dynamic loading). These figures are not very different from those that would be expected if the application had been coded in C++.

CPU usage measurements were performed on each thread (see description of scheduling policy above). The execution time for one single cycle for each thread was computed and the results are summarized in Table 7.2.
7.6. Instantiated Control Applications

<table>
<thead>
<tr>
<th>Task</th>
<th>Measured Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Periodic Task 1</td>
<td>10-15 ms during normal operations, peaks of up to 26 ms during manoeuvre executions and when there is mode change</td>
</tr>
<tr>
<td>Periodic Task 2</td>
<td>1 ms</td>
</tr>
<tr>
<td>Telemetry Thread</td>
<td>10-17 ms depending on the amount of telemetry data</td>
</tr>
<tr>
<td>Telecommand Thread</td>
<td>1 ms</td>
</tr>
</tbody>
</table>

Table 7.2: CPU measurements of the control application deployed on the Jbed & PowerPC platform.

Note that there are no tools for computing the worst case execution times of a Java applications and the above values were derived from observation of a large number of measurements.

7.6.2 The Control Application for the Swing Mass Model Deployed on the PERC Platform

This section describes the second version of the control application for the swing mass model (SMM) that differs from the first version primarily because it is built upon a different real-time Java implementation (PERC instead of Jbed), but also because it assumes a slightly more complex SMM configuration (right-hand motor of the SMM system is used to generate a disturbance torque) and because its scheduling approach is different (five threads instead of four).

The control application was instantiated only by the factory instantiation approach described in Section 7.7.2. An overview of the control application instantiated from the RTJ Framework for the PERC & VxWorks & PC platform (described in Section 7.5.2) is presented below.

7.6.2.1 Description of the Control Application

The control application is described in terms of configuration actions performed on particular components in order to achieve the required behavior of the control application. The description can found in Section A.2, Appendix A.

7.6.2.2 Scheduling Approach

The prototype application uses the following framework active components:

- telecommand loader
- telecommand manager
- controller manager
- failure detection manager
failure recovery manager
manoeuvre manager
unit manager
telemetry manager

Except for the unit manager, all other active components are instantiated only once. Two instances of the unit manager are instead used. The first instance – the sensor unit manager – is used to acquire the measurements from the application sensors. The second instance – the actuator unit manager – is used to send the actuation commands to the application actuator and to the disturbance actuator.

The control cycle for the prototype application has a fixed period with duration of 200 ms. The only scheduling constraints that apply to the scheduling of the above active components are as follows:

- the sensor unit manager must be triggered at the beginning of a cycle and must terminate execution within 10 ms from the beginning of the cycle (this is necessary to ensure that the sensor measurements are collected sufficiently early in the cycle)
- the actuator unit manager must be triggered not earlier than 30 ms into the cycle (this is necessary to ensure that the actuator is stimulated sufficiently late into the cycle).

In view of the above and in view of the capabilities of the PERC run-time system, the scheduling approach selected for the prototype application is articulated over the following five real-time threads:

- *Periodic Task 1*
- *Periodic Task 2*
- *Periodic Task 3*
- *Telemetry Thread*
- *Telecommand Thread*

All five tasks have fixed priorities: the *Periodic Tasks* have the priority received from a timer with which they are implemented. The *Telemetry Thread* has priority 7 and *Telecommand Thread* has priority 6. It is assumed that the *Periodic Tasks* have all priority 10 since their priorities can not be set and the applications runs properly also with higher priorities for *Telemetry Thread* and *Telecommand Thread* but there is no proof of it. The PERC run-time system schedules threads using a fixed priority scheduler with pre-emption and priority inheritance. For equal priorities round robin scheduling is used. Since the priority level of 10 is the highest in the PERC system it ensures that a thread is only preempted by other threads of priority of 10. Note that on VxWorks round robin can be disabled for the PERC run-time system.

The basic idea of the proposed scheduling scheme is that all operations that require timing
determinism are performed in the (high priority) *Periodic Tasks*, implemented as timer tasks at fixed rate in a timer, whereas the (lower priority) *Telemetry Thread* and *Telecommand Thread* is used for operations that are not real-time safe. In particular, the management of the FTP socket that supports the telemetry/telecommand (TM/TC) link is performed in the *Telemetry Thread* and *Telecommand Thread*. In order to ensure that there is no disruption of the scheduling of the *Periodic Tasks*, the interaction between them and the *Telemetry Thread* and *Telecommand Thread* is kept to a minimum and is tightly controlled to ensure that potentially blocking calls that are performed from the *Telemetry Thread* and *Telecommand Thread* on objects that are used by the *Periodic Tasks* have a predictable and limited blocking time.

The *Periodic Task 1* is released first and triggers the sensor unit manager component.

The *Periodic Task 2* defines the control cycle for the prototype application. Its main function is to trigger the following framework active components (in the order in which they are listed):

1. telecommand manager
2. controller manager
3. failure detection manager
4. failure recovery manager
5. manoeuvre manager
6. telemetry manager

The *Periodic Task 3* is triggered by the timer as the last task every period. Its function is to trigger the actuator unit manager component.

The triggering of all *Periodic Tasks* is done every period by the timer.

The *Telemetry Thread* runs with a lower priority than the periodic tasks and its function is to send the telemetry buffer to the remote operator through the FTP socket. This is done by activating the telemetry sender active component (see telemetry policy in Section A.2.6 of Appendix A). Ideally, the *Telemetry Thread* should be activated immediately after the *Periodic Task 2* has completed and should complete before *Periodic Task 3* begins. This would ensure an orderly forwarding of telemetry data: the telemetry manager (which is activated by *Periodic Task 2*) fills in a telemetry buffer which is then emptied by the *Telemetry Thread*. This ideal case is illustrated in Figure 7.9.

However, the *Telemetry Thread* performs two operations that are not real-time safe:

- it forwards telemetry data over a data network that is shared with other users and where timeliness of service is not guaranteed
- it performs dynamic memory allocation operations because when a telemetry buffer is retrieved from the telemetry stream component for flushing, a new, empty, buffer must be created and supplied to the telemetry stream
The possibility therefore exists that the telemetry thread will not finish before the Periodic Task 3 is ready to start. In such a case, the PERC scheduling mechanism ensures that the Telemetry Thread is preempted and that Periodic Task 3 is executed. The Telemetry Thread will resume execution as soon as possible. This may cause one telemetry buffer not be forwarded to the remote operator (loss of one frame of telemetry).

The synchronization between the Periodic Tasks and the Telemetry Thread is performed as follows. The Telemetry Thread goes to sleep on a call to wait after it has forwarded one telemetry frame. It is awoken by Periodic Task 2 that sends it a notify signal after executing all its functionality managers. The Telecommand Thread runs with the lowest priority and can be preempted by every other task. Its function is to collect telecommands from the FTP socket. Whenever it is awoken, it checks whether a new telecommand has been received and, if so, it loads it into the telecommand loader (see the telecommand policy in Section A.2.5 of Appendix A). The telecommand loader will dynamically create an object to encapsulate the telecommand and will then load it into the telecommand manager. The Telecommand Thread then goes to sleep for a period of 200 ms. The Telecommand Thread runs completely asynchronously from all other application threads. If the remote operator sends two telecommands in between two successive awakenings of the telecommand receiver thread, the earlier of the two telecommands will be lost. Processing of telecommands could be made more deterministic by linking its release to the end of Periodic Task 2 and by raising its priority to the same level as the level of Periodic Tasks.

Note that the PERC run-time system creates an additional thread to run the garbage collector which priority can also be set while the garbage collector is not disabled in the PERC VM. This thread normally runs in the background with the lowest priority. However, if a task makes a request for a memory allocation that cannot be serviced due to a lack of free memory, the garbage collector is invoked with a priority equal to that of the calling task. Finally, when PERC VM is running under the control of a remote terminal (as would be the case for testing), two additional threads are created by the run-time system to manage the debugger connection to the remote terminal.
Finally, it has to be stressed that the implementation proposed above relies on the Timer class for the Periodic Tasks and the standard Java sleep service for the other threads. The Periodic Tasks running on the timer are not preempted using round robin and the triggering time of single tasks can only be called "around" the wanted start time but there is all in all no time shift.

Note that PERC provides no other constructs to implement periodic threads, Java provides it by the Thread class and the Timer class, but it has to be emphasized that there is a special sleep service provided by PERC using milli- and nanoseconds as parameters. This service is not used for the Telecommand Thread since there is no such time accuracy needed for this thread.

7.6.2.3 Memory and CPU Measurements

Memory measurements were performed on the instantiated control application for the SMM (see Section 7.4.1) deployed on the PERC & VxWorks & PC platform (see Section 7.5.2) in its final form. The CPU usage measurements were performed on a special measurement application because of the high processor frequency and the low accuracy of the system time. The memory measurements are summarized in Table 7.3. The table compares the measurements for the SMM control applications using PERC with the measurements for the standard “Hello World” program and a “naked” PVM. The figures in Table 7.3 give total memory usage including both code and data.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>SMM control application</th>
<th>Hello World</th>
<th>naked “PVM”</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory with static linking of native methods</td>
<td>10937 KB</td>
<td>9445.3 KB</td>
<td>9444 KB</td>
</tr>
</tbody>
</table>

Table 7.3: Memory measurements of the SMM control application and “Hello World” program deployed on the PERC & VxWorks & PC platform.

The Java execution model implemented by PERC is described in Section 3.3.1.2. The Java class files are ROMized to an image file and linked with the JVM. Before generating the image, the ROMizer compiles the class files into native code. There are two possibilities to link the native methods. One is to link it statically into the executable module or to load them directly into VxWorks and link them dynamically. Since there is no data storage system used only static linking was available.

The figures in the table indicate that the basic PERC run-time system takes about 9 MBytes. An interesting observation can be made when linking the executable module (application + PVM) directly into VxWorks while generating the VxWorks image instead of loading it during runtime. Then the whole system (application + PVM + VxWorks) occupies 8.12 MB. The application specific code for the prototype SMM application takes a little less than 1.5 MB when loading it during runtime.
There are currently no tools for computing the worst case execution times of Java applications provided by PERC. Therefore CPU usage measurements required to develop a special measurement application using most of the classes used in the original application. There are only two new classes used: 
\texttt{TimeVxPercSmm\_1} is used instead of \texttt{PercSmmTimer\_1} as the main class and also a special instantiator class is used. Note that changes only apply for the time measurement experiments. The measurement application performs measurements for every single manager functionality with the exception that failure detection and failure recovery are performed together because of their strong connection. Measurements were performed for the idle and the worst case by calling the manager as an application parameter. In most cases a second parameter has to be provided, which describes the number of manager calls. In certain cases the number of iterations is not needed or the connection to the remote operator is demanded. The results of all worst case measurements of the tasks are listed in Table 7.4.

<table>
<thead>
<tr>
<th>Task</th>
<th>Measured Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Periodic Task 1</td>
<td>75.1 (\mu s)</td>
</tr>
<tr>
<td>Periodic Task 2</td>
<td>333.4 (\mu s)</td>
</tr>
<tr>
<td>Periodic Task 3</td>
<td>26.8 (\mu s)</td>
</tr>
<tr>
<td>Telemetry Thread</td>
<td>800 (\mu s)</td>
</tr>
<tr>
<td>Telecommand Thread</td>
<td>50 (\mu s)</td>
</tr>
</tbody>
</table>

Table 7.4: CPU measurements of the control application deployed on the PERC & VxWorks & PC platform.

Note that the worst case measurements of \textit{PeriodicTask 2} are summarized of the worst case measurements of all its managers.

### 7.6.3 The Control Application for the ER1 Robot Deployed on the Java Platform

The third instantiated control application differs from the previous two in several aspects. First, the goal is to navigate the ER1 robot to a defined position. Second, a standard Java implementation was used instead of specialized real-time Java implementations. Third, a personal computer was used as a hardware platform. On the other hand, the requirements on the control application were not as complex as in the control applications for the swing mass model. The main objective was to develop an application that will control conceptually different system than the one used in the previous cases where the main focus was on testing framework functionality and measuring performance data.

The control application was instantiated by two different instantiation approaches: the factory instantiation approach described in Section 7.7.3 and the automated instantiation approach...
described in Section 7.8.2.

7.6.3.1 Description of the Control Application

The control application is described in terms of configuration actions performed on particular components in order to achieve the required behavior of the control application. The description can be found in Section A.3, Appendix A.

7.6.3.2 Scheduling Approach

The robot control application uses the following active components:

- telecommand loader
- telecommand manager
- controller manager
- manoeuvre manager
- unit manager
- telemetry manager

As in the previous two applications, the robot control application contains two instances of the unit manager, all other components are instantiated only once. The first instance of the unit manager is used to acquire the measurements from the robot position sensor. The second instance of the unit manager is used to send the actuation commands to the stepper motors.

The control cycle for the prototype application has a fixed period with duration of 1 second. It is important to stress that the robot control application is a non-real-time application deployed on standard Java. Therefore, the active components are managed by four standard Java threads:

- Periodic Thread 1
- Periodic Thread 2
- Telemetry Thread
- Telecommand Thread

All four threads have fixed priorities: Periodic Thread 1, Periodic Thread 2, and Telemetry Thread have priority 10, Telecommand Thread has priority 5. Although the priority level 10 is the highest in standard Java, there is no guarantee that a thread will not be preempted by a thread with a lower priority level.

The Periodic Thread 1 is responsible for triggering the following active components (in the order in which they are listed):

1. sensor unit manager
2. telecommand manager  
3. controller manager  
4. manoeuvre manager  
5. telemetry manager

When the listed managers complete their execution, then the Periodic Thread 1 goes to sleep. The Periodic Thread 2 runs with a period of 1 second, but is released 500 ms after the Periodic Thread 1 is released. The Periodic Thread 2 triggers only the actuator unit management component. When the execution of the actuator unit management component is finished then the thread goes sleep until the next invocation, which is determined by the next period.

The Telemetry Thread has the same priority as Periodic Thread 1 and Periodic Thread 2, which means that the sending telemetry data should not be preempted by the threads executing functionality managers. Its responsibility is to send the telemetry data to the remote operator through the FTP socket. The Telemetry Thread should be activated after the Periodic Thread 1 has finished, in fact, the Periodic Thread 1 notifies the Telemetry Thread after executing all its functionality managers.

The Telecommand Thread is executed with a lower priority than the Periodic Thread 1, Periodic Thread 2, and Telemetry Thread. Its function is identical as in two previous applications (see Section 7.6.1 and 7.6.2) – it is processing telecommands from the FTP socket. First, it checks whether a new telecommand has been received. Second, it loads a telecommand into the telecommand loader. Third, the telecommand loader dynamically creates an object to encapsulate the telecommand and will then load it into the telecommand manager that executes the telecommand. Fourth, the Telecommand Thread goes to sleep for a period of 1 second. The implemented scheduling approach is illustrated in Figure 7.10.

![Figure 7.10: The scheduling policy of the robot control application.](image)

### 7.6.3.3 Performance

The standard Java & PC platform (see Section 7.5.3) was the most powerful in comparison to other two (described in Section 7.5.1 and 7.5.2). Nevertheless, the Periodic Thread 1 and Periodic Thread 2 mostly did not complete execution of their active components at a reasonable time. This was due to the fact that both contain a unit active component. The unit active
component in the Periodic Thread 1 acquires position data, in the Periodic Thread 2 sends command data to the stepper motors. Since a direct access to the sensors and from the actuators is not possible, the communication between the control application and robot hardware goes through Robot Control Center (RCC) software that is used to control and receive a feedback from the ER1 robot. In addition to that, each data acquisition and actuator command sent to RCC is confirmed by an acknowledgement message when the command is executed. Naturally, this produces enormous delays that very often result in a situation that the high priority thread is waiting for a response from the ER1 robot and blocks other threads at the same time. Therefore, the measurement of the execution time of particular threads was not performed.

The memory measurements are summarized in Table 7.5. The table provides figures of the robot control application and standard “Hello World” program. Both applications were executed by Java 2 Runtime Environment (build 1.4.1_02-b06), and measured by JProbe performance toolkit for Java [70].

<table>
<thead>
<tr>
<th>Measured Item</th>
<th>Robot Control Application</th>
<th>Hello World</th>
</tr>
</thead>
<tbody>
<tr>
<td>used memory</td>
<td>607 KB</td>
<td>396 KB</td>
</tr>
</tbody>
</table>

Table 7.5: Memory measurement of the robot control application and “Hello World” program deployed on the Java & PC platform.

### 7.7 Factory Instantiation Approach

The factory instantiation approach described in Section 4.4.2 was applied to instantiate three different control applications (see Section 7.6) deployed on three different platforms (see Section 7.5).

The factory instantiation approach requires development of application specific factories that extend the provided framework factories. The application factories represent a generic implementation of a configuration knowledge for a limited set of applications. Further specialization depends on an implementation of application specific components and configuration files setting component properties. The instantiation and configuration of the application factories and components is performed in a dedicated instantiator component module.

#### 7.7.1 The Control Application for the Swing Mass Model Deployed on the Jbed Platform

The control application for the swing mass model deployed on the Jbed & PowerPC platform is organized as a single package called JbedSmmApplication. The package structure is similar to a structure of the framework packages that provides framework code. For example, the framework package RtiAocsFramework has a sub-package AocsData that contains
the classes required to support the data concept. The package `JbedSmmApplication` has a sub-package with the same name that contains the classes that implement the application specific data pool concept for the SMM application.

### 7.7.1.1 Test and Target Application

Based on the proposed testing concept for the framework (see Section 3.4.6), two executable applications were instantiated. The first executable application is used for the system-level testing in the desktop environment. Its main class is `TestSmm_1`. The second executable is the application to be downloaded to the Jbed target. Its main class is `JbedSmm_1`. These two applications are kept as similar as possible. The differences are summarized in Table 7.6.

<table>
<thead>
<tr>
<th>Functionality</th>
<th>Test Application</th>
<th>Jbed Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>time reference</td>
<td>The <code>TestSmmClock</code> class that relies on the standard Java timer class to retrieve time information.</td>
<td>The <code>JbedSmmClock</code> class represents the on-board clock. This classes uses a Jbed timer.</td>
</tr>
<tr>
<td>data acquisition</td>
<td>The <code>TestSmmAnalogSensor</code> class simulates a sensor that supplies a constant sensor reading.</td>
<td>The <code>JbedSmmAnalogSensor</code> class encapsulates the interface to the SMM sensor.</td>
</tr>
<tr>
<td>performing of actions</td>
<td>The <code>TestSmmAnalogActuator</code> class simulates an actuator that writes its command to an external file.</td>
<td>The <code>JbedSmmAnalogActuator</code> class encapsulates the interface to the SMM actuator.</td>
</tr>
</tbody>
</table>

Table 7.6: The differences between the test application deployed on a desktop and the target control application for the SMM deployed on the Jbed & PowerPC platform.

### 7.7.1.2 Application Specific Factories

Twelve application factories were developed to assist the instantiation of the control application for the SMM deployed on the Jbed & PowerPC platform, see Section B.1.1 from Appendix B.

### 7.7.1.3 Application Specific Components

The final customization of the control application is driven by application specific components and their configuration. Therefore, several application specific components and the configuration component were developed to satisfy given specifications of the control application for the SMM deployed on the Jbed & PowerPC platform. The overview of the application components can be found in Section B.1.2 that is a part of Appendix B.
7.7. Factory Instantiation Approach

7.7.1.4 Instantiation Process

The factory approach is based on the development of a set of abstract and application specific factories that are responsible for a construction and configuration of framework and application specific components. In the case of the control application for the SMM deployed on the Jbed & PowerPC platform it was the instantiation and configuration of application specific factories by application specific components performed in the JbedSmmInstantiator component. The application used for the system-level testing in the desktop environment is instantiated in the TestSmmInstantiator component.

7.7.2 The Control Application for the Swing Mass Model Deployed on the PERC Platform

The control application for the SMM deployed on the PERC & VxWorks & PC platform is stored in a single package called PercSmmApplication. The structure of this package mirrors that of the RjAocsFramework package that contains the framework code.

7.7.2.1 Test and Target Application

In accordance with the testing concept proposed for the framework (see Section 3.4.6), two application executables can be generated from the PercSmmApplication. The first one is the executable to be downloaded to the PERC target. Its main class is PercSmmTimer_1 (there is also an executable to be downloaded to the PERC target where all tasks are implemented by threads instead of using timers. It is called PercSmm_1. The second executable is used for the system-level testing in the desktop environment. Its main class is TestSmm_1. These two applications are kept as similar as possible. The only differences are summarized in Table 7.7.

7.7.2.2 Application Specific Factories

Fourteen application factories were developed to assist the instantiation of the control application for the SMM deployed on the PERC & VxWorks & PC platform. Their description can be found in Section B.2.1 of Appendix B.

7.7.2.3 Application Specific Components

Although the application specific factories that instantiate the control application deployed on the PERC & VxWorks & PC platform perform similar instantiation actions as the application specific factories for the Jbed & PowerPC platform, the final application is very different. This is due to the fact that different application specific components were developed and instantiated to complete the control application according to given specifications. The application specific components and the configuration component used to instantiate the control application for the SMM deployed on the PERC & VxWorks & PC platform can be found in Section B.2.2 of Appendix B.
Table 7.7: The differences between the test application deployed on a desktop and the target control application for the SMM deployed on the PERC & VxWorks & PC platform.

<table>
<thead>
<tr>
<th>Functionality</th>
<th>Test Application</th>
<th>PERC Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>time reference</td>
<td>The TestSmmClock class that relies on the standard Java timer class to retrieve time information.</td>
<td>The PercSmmClock class represents the on-board clock. This classes uses a PERC timer.</td>
</tr>
<tr>
<td>data acquisition</td>
<td>The TestSmmAnalogSensor class simulates a sensor that supplies a constant sensor reading.</td>
<td>The PercSmmAnalogSensor class encapsulates the interface to the SMM sensor.</td>
</tr>
<tr>
<td>performing of actions</td>
<td>The TestSmmAnalogActuator class simulates an actuator that writes its command to an external file.</td>
<td>The PercSmmAnalogActuator class encapsulates the interface to the SMM actuator.</td>
</tr>
</tbody>
</table>

7.7.2.4 Instantiation Process

Instantiation factories are components that are responsible for creating and configuring a set of components. Three different components performing their instantiation and configuration were developed:

- PercSmmInstantiator – is the instantiator component of the control application for the SMM deployed on the PERC & VxWorks & PC platform.

- TestSmmInstantiator – is the instantiator component of the control application for the SMM deployed on the desktop.

- TimePercSmmInstantiator – is the instantiator component of the control application for the SMM performing time measurement tests on Windows.

7.7.3 The Control Application for the ER1 Robot Deployed on the Java Platform

The control application for the ER1 robot deployed on the standard Java & PC platform (see Section 7.5.3) is stored in a single package called RobotApplication. The structure of this package reflects the structure of the RtjAocsFramework package with the framework code.
7.8. Automated Instantiation Approach

7.7.3.1 Test and Target Application

The only possible target platform is a personal computer with Windows operating system and standard JVM. Therefore only one executable control application is instantiated. The control application is used for both the system-level testing and for the final deployment.

7.7.3.2 Application Specific Factories

Ten application factories were developed (see Section B.3.1 in Appendix B) to assist the instantiation of the control application for the ER1 robot deployed on the standard Java & PC platform (see Section 7.5.3).

7.7.3.3 Application Specific Components

To instantiate a control application for the ER1 robot required developing a new set of application specific components, see Section B.3.2 in Appendix B. Most of the new components are feasible only for the robot control application, and their reuse for other applications is not possible. On the other hand, the robot control application uses some components that were initially implemented for the control application of the SMM. Since these components were used in all three applications they could become the framework default components.

7.7.3.4 Instantiation Process

As in the two previous control applications instantiated using the factory approach the instantiation process of the control application for the ER1 robot was also deployed on the Java & PC platform was carried out by application specific factories and components instantiated and configured by the RobotInstantiator instantiator component.

7.8 Automated Instantiation Approach

The automated instantiation approach described in Section 5.2 was applied to instantiate two different control applications deployed on two different platforms.

The objectives to instantiate control applications using the automated instantiation approach are to demonstrate the innovative instantiation technique, its effectiveness and feasibility for object oriented frameworks compliant with design and implementation conventions discussed in Section 2.6, and also demonstrate reusability and flexibility of the RTJ Framework.

7.8.1 The Control Application for the Swing Mass Model Deployed on the Jbed Platform

The instantiated application by the automated instantiation approach is the same application that was instantiated by the factory approach (see Section 7.7.1). The factory approach was
based on the development of a set of abstract factories that constructed and partially configured the components required for the application. The configuration of these components and their assembly was then performed in a dedicated instantiator component module. A detail description of the control application can be found in Section 7.6.1.

7.8.1.1 Preparation Steps

The automated instantiation approach assumes that all the components required for the application are already available. If not, then a developer has to focus on their development first. In our case, the components have already been developed for the instantiation of the control application by the factory approach (see Section 7.7.1.3). Components have to satisfy certain conditions, such as: naming conventions, instantiation and configuration actions must be expressed in terms of a small set of instantiation operations, etc. These conditions are satisfied by all RTJ Framework components and by specific components obtained by specializing framework components. Prepared components can then be imported to the Graphical Instantiation Environment (see Section 5.2.1). This is easy to do as it simply required the palette file of the customized Bean Builder to be edited with the names of the framework and application specific components.

7.8.1.2 Instantiation Process

The instantiation of the control application for the SMM deployed on the Jbed & PC platform was performed entirely within the customized Bean Builder. The application components were pulled down from the component palette and configured using the configuration wizards and the property sheets provided by the environment. The total number of components used for the application was 75. Figure 7.11 shows the control application instantiated in the customized Bean Builder. The time required to fully construct the application was in the order of a few hours. The instantiation was done in several stages. The instantiation process was broken up in stages because partial configurations could be saved and restored using the save/restore commands of the customized Bean Builder. The simulation facilities (in particular the configuration check) provided by the customized Bean Builder played a very important role in speeding up the instantiation process. The designer could check the completeness of the configuration process by performing periodic configuration checks that gave valuable indications which components still required attention. The time required for a complete configuration check is in the order of seconds which allowed this facility to be used frequently. After all components required for the application were instantiated and their configuration was completed (completion was approved by running the configuration check), the application configuration file was automatically generated. Finally the instantiation code was generated. The instantiation class was integrated and linked with the Jbed operating system to generate the final executable module. The first time it was run, it found an error, which was due to a component that was not completely configured. The incomplete configuration was in turn not detected by the configuration check owing to a bug in the implementation of the configuration
7.8. Automated Instantiation Approach

check for a particular component. No other problems were encountered. This confirms that, assuming that the individual components are bug-free, the instantiation environment can be used to automatically and directly generate executable applications that are ready for use.

Figure 7.11: The instantiated control application for the SMM deployed on the Jbed & PowerPC platform developed in the customized Bean Builder.

7.8.2 The Control Application for the ER1 Robot Deployed on the Java Platform

The instantiated control application is identical with the application instantiated by the factory approach that is described in Section 7.7.3. A detail description of the control application can be found in Section 7.6.3.
7.8.2.1 Preparation Steps

To be compliant with requirements on the automated instantiation approach, at first the developer must provide all components that participate during the instantiation process. In fact, all required components have already been developed for the instantiation of the control application by the factory approach (see Section 7.7.3). However, the components have to be transformed to descriptor models before they can be imported to the generic Graphical Instantiation Environment (GIE) that plays a central role in the automated instantiation approach, see Section 5.3. The description model is defined as a set of Java interfaces, it consists of a component and connection description model and a configuration knowledge.

The transformation of the components to their description models can be either done manually or performed automatically. The automatic transformation would be usually preferable. Therefore, a generic XSL program was developed to transform the framework and application specific application components to a set of descriptors required by GIE. The transformation process is described in Section 5.3.

The required deployment format of the component description models into GIE is a jar file containing their compiled classes together with all other resources (e.g. icons of component, connections, etc.). In addition to that, a GIE file including a link to the description has to be provided.

7.8.2.2 Instantiation Process

The instantiation process of the control application for the ER1 robot deployed on the Java & PC platform was divided into two stages. In the first stage the control application was instantiated in the user friendly GIE tool. The second stage focuses on a transformation of instantiated operations encoded to a XML document to instantiated operations at the level of the application language (in the case of the RTJ Framework it is Java).

The component instantiation and configuration was performed only within the GIE tool. The tool provides a palette with framework and application components, and connections. Component instantiation is a simple operation – a developer selects a component from the palette and then clicks on a workspace area. Linking components is done by selecting an appropriate connection type from the palette clicking on the first component and then on the second. Component properties are displayed in a special property window where they can be further customized. The instantiated operations are stored to a XML document, which is also used for restoring of the application.

The control application for the ER1 robot consists of 53 instantiated components. The screenshot of the GIE tool shown in Figure 7.12 shows the finalized control application for the ER1 robot.

Currently GIE does not provide any concept for validation of instantiated applications. Therefore was not possible to perform the configuration check or other type of simulation, as it was possible within the customized Bean Builder.

When the application is completed in the GIE tool, then the application configuration XML file is automatically transformed to an instantiation code. This step was not practically imple-
mented, because conceptually it is the same type of a transformation that was already tested in the customized Bean Builder tool.

Figure 7.12: The instantiated control application for the ERI robot deployed on the Java platform developed in the GIE.

7.9 Feature-Based Instantiation Approach

The feature instantiation approach was applied to instantiate the control application for the SMM deployed on the Jbed & PowerPC platform.

The objective of the feature instantiation approach is different from the factory and automated instantiation approaches. Although all three approaches have the same goal – to generate or provide customizable instantiation code – only the instantiation code implemented by the factory and automated instantiation approaches instantiate the whole application. In the case of the feature instantiation approach, the generated code is usually incomplete and therefore must be further customized in component composition environments (e.g. the customized Bean-Builder or GIE).
The feature instantiation approach (described in detail in Section 6.3) is based on the feature modelling technique for software frameworks introduced in Section 6.2.

### 7.9.1 The Control Application for the Swing Mass Model Deployed on the Jbed Platform

The description of the instantiated control application can be found in Section 7.6.1.

#### 7.9.1.1 Preparation Steps

The first step is the development of a framework feature model. Usually this is done during the domain analysis phase. However, in case of the RTJ Framework the framework feature model was defined additionally.

A result of the instantiation process depends on a type and completeness of information associated to particular features in the framework feature model. A software framework is organized as a set of building blocks that need to be configured to form the required target application. The framework feature model has to contain description of relations between features and the framework building blocks. The concept for expressing the relationship between features and the framework building blocks is described in Section 6.3.2. The XML code below illustrates one type of possible relations. It is mapping of a feature to a component. In particular, the feature “Controller” is linked to the Controller component stored in the RtjAocsFramework.ControllerManagement package.

```xml
<Feature nameOfFeature="Controller">
  <Description text="A feature that covers functionality of a SISO controller.="/>
  <FeatureCardinality FeatureCardMin="1"
                      FeatureCardMax="+="/>
  <Type>
    <BuildingBlock>
      <Component status="provided" type="RtjAocsFramework.ControllerManagement.Controller"/>
    </BuildingBlock>
  </Type>
</Feature>
```
Compared to the automated instantiation approach, the feature modelling approach does not require all components necessary for the target application to be available. The feature modelling concept classifies component availability to three categories: as provided, generable or missing. Components that are generable and missing do not exist, but the implementation of generable components can be automatically generated, in case of missing a skeleton is generated that has to be completed manually.

All components required by the control application for the SMM were already available. They were either provided by the framework or developed in the previous instantiation experiments. The application feature model is stored in a single XML file. The obtained application model is used to automatically identify and instantiate components required for a target application. Therefore, a framework specific XSL-based code generator had to be developed to perform this kind of a transformation.

### 7.9.1.2 Instantiation Process

An instantiation process performed by the feature instantiation approach can be seen as pruning the framework feature model to obtain an application model that represents specifications of the target applications. Since the framework feature model is encoded in a XML document, a low cost XML editor (oXygen [81]) was found satisfactory for this type of operations. oXygen offers facilities to automatically enforce compliance with the meta-models implemented as XML Schemas. Another possibility would be to use a graphical feature modelling tool that works with the proposed modelling approach requiring three modelling levels: the framework meta-modelling level, framework modelling level, and application modelling level. A prototype version of the tool is described in [89], unfortunately the tool is not mature enough to instantiate application model according to the proposed concept.

After the application model was defined by pruning the framework feature model, a XSL program is used to transform the application model containing only features required by the control application to a file containing a set of unconfigured components that are supposed to form the control application. This file was then imported to the customized component composition Bean Builder tool (see Section 5.2.1) for further configuration of the instantiated components.

### 7.10 Conclusions

Five general conclusions can be drawn from the development experience of control applications instantiated from the RTJ Framework.

First, the framework technology can and should be used in the control domain. The RTJ Framework was tested by using it to develop control applications for two plants deployed on three different platforms. The results obtained during the instantiation process are encouraging and confirmed that the RTJ Framework is easily adaptable to different type of control applications and platforms. Two-thirds of classes in the final applications were provided by the framework. The development of the remaining classes was facilitated by the need to adhere to
the framework design patterns and abstract interfaces.
The second conclusion is supported by the experience of deploying the control applications to the real-time versions of Java. Within the limitations of the selected platforms – Esmertec’s Jbed and NewMonics’s PERC – the deployment phase was successful. Java is becoming a realistic option for real-time control systems, but a caution is required in selecting the run-time system.

Third, the presence of the Java Virtual Machine (JVM) was the chief advantage of using Java, because it blurs the distinction between the target embedded system and the desktop environment. The “write-once-run-anywhere” (WORA) claim was confirmed based on experiences acquired during the instantiation and deployment phases even for the real-time control domain.

Fourth, the new automated instantiation approach proved to be used to assemble a full application exclusively through graphical means and to generate the final application code with no need to write any code.

Fifth, the feature-based modelling and instantiation was tested on the RTJ Framework. The application specification process was found to be very convenient for users (application developers). This is primarily because the application meta-model that is generated from the framework model is very constrained and, at each step of the application definition process, it provides a narrow list of choices. This simplifies the specification job, reduces errors and significantly speeds up the development process.
CHAPTER 8

Conclusion

This chapter sums up the contribution of this work, discusses the achieved goals, highlights the main contributions, and also points out suggested areas of further work.

Contents

8.1 Summary ......................................................... 142
8.2 Achieved Goals ............................................... 142
8.3 Main Contributions .......................................... 143
8.4 Future Work .................................................... 144
8.1 Summary

This thesis was originally motivated by absence of a generic concept for framework instantiation.

The work started more or less “from scratch”, i.e. it was necessary to re-engineer a software framework that could be used as a test bed (Chapter 2). A software framework is ideally suited to define the overall architecture for control applications and provide an umbrella under which components coming from commercial autocoding tools (suitably wrapped) can be combined with each other and with components coming from other sources to build the final application. The assessment is that software frameworks can bring the same benefits to the control system domain that they have brought to other disciplines.

The other investigated issue was the use of the Java language as an implementation technology for the software frameworks covering the domain of real-time control systems (Chapter 3). There are currently several well-established solutions for real-time Java. Some of these are mature enough for use in control systems. The designed RTJ Framework for control systems was implemented in Java because of its attractive safety features and real-time implementations that are of interest in control applications. The RTJ Framework was successfully tested on two real-time Java implementations, Esmertec’s Jbed and NewMonics’s PERC. The chief advantage of using Java for programming of control systems is the presence of the Java Virtual Machine that insulates the application from the underlying platform and from the RTOS. This allowed nearly all of the development to be done on a convenient desktop environment. The experience certainly confirmed that Java’s claim “write-once-run-anywhere” holds for control systems too.

A software framework can also be seen as complementary to a Matlab-based approach. The appeal of the Matlab-based approach lies in its GUI-oriented user interface that allows the control software to be developed directly by the control engineer with only minimal assistance from a software engineer. Naturally, a framework-based approach will only be accepted in the control community if it can be packaged in a similar way. This is exactly what the new automated and feature-based instantiation approaches (Chapter 5 and 6) are trying to offer. The distinctive feature of these approaches is their simplicity and reliance only on mainstream technologies. This would allow spreading the new instantiation concepts to reach of most framework designers and users.

In conclusion, the developed RTJ Framework and the proposed instantiation approaches were tested to instantiate control applications for two laboratory models deployed on three different platforms.

8.2 Achieved Goals

The main objective of this work was to develop a concept that facilitates the framework instantiation process. This task was at the very beginning divided into several parts, and the work is believed to achieve all defined goals in Section 1.4:
8.3 Main Contributions

There are five main contributions of this thesis to the control and software engineering domain:

1. The existing AOCS Framework [66] was redesigned, improved and reimplemented. The whole process resulted in the RTJ Framework, an object-oriented software framework implemented in Java language that has an ambition to cover the domain of control systems.

2. An evaluation of suitability of Java language for real-time control applications based on an object-oriented software framework was done. The evaluation study discusses important issues related to the control domain. The RTJ Framework presents a practical experience of implementing an object-oriented framework in a real-time version of Java.

3. A new automated instantiation approach that facilitates the framework instantiation process was proposed and implemented. The central part of this concept is a GUI-based tool where control specialists can intuitively develop and simulate a control application based on a software framework, and finally automatically generate its instantiation code.

4. The feature-based modelling technique was extended and improved in several respects. In particular, it introduces a concept of encapsulating features in macro modules that...
can be independently applied. It also proposes a new mechanism for global composition constraints using XPath allowing expression of any generic constraint based upon the combination of features and their values. It defines an XML-based approach for expressing the feature models that offers a low-cost path to the development of a support tool for building the models.

5. A new feature-based instantiation concept that allows instantiation of the target application components directly from an application feature model and performs simple configuration operations was proposed and implemented.

In addition, other contributions of this thesis are control applications for two plants deployed on three platforms instantiated by the new instantiation approaches.

8.4 Future Work

The future work will follow two broad directions. On the one hand, the objective is to test the proposed instantiation methods by applying them to industrial test cases, since the experience to date is restricted to laboratory experiments (see Chapter 7). The obtained results were encouraging enough to try a more ambitious experiment, porting the concepts described in Chapter 5 and 6 to a new framework and applying it to the generation of operational software in an industrial context. On the other hand, in a more research-oriented line of work, there is a possibility of extending the capability of the instantiation environment to perform component customization as well as component configuration. A component is configured by acting upon it through the operations it declares in its external interface. A component is customized if its internal implementation is modified. The code customization will require the use of aspect oriented programming techniques. In particular, an idea would be to explore whether the problem of customizing can be transformed to a set of components with respect to a certain feature into an equivalent problem of configuring a meta-component that describes that feature. This would allow us to preserve much of the current approach that is based on component configuration with the important difference that some of the components that are configured in the environment would be meta-components (or maybe their visual proxies) and that the code that is generated from them is component customization code rather than application implementation code.

Results of this work are used in several internal projects of Automatic Control Laboratory, ETH Zürich.

More information can be found at: http://control.ee.ethz.ch/~cechti
A.1 The Control Application for the Swing Mass Model Deployed on the Jbed Platform

A.1.1 Operational Modes

The following components in the prototype application are mode-dependent:

• failure detection manager
• failure recovery manager
• controller manager
• telemetry manager

The first three components have the following two modes:

• stand-by mode ($SBY$)
• run mode ($RUN$)

Their modes are slaved to the mode of a mission mode manager component. This ensures that their mode changes are synchronized. The $SBY$ mode is the default mode entered immediately after start-up. In $SBY$, the application processes telecommands and generates telemetry. There is also no closed-loop control of the swing mass speed.

The telemetry manager has the following two modes:
• telemetry format 1 (TM_FMT1)
• telemetry format 2 (TM_FMT2)

The telemetry manager cyclically alternates between these two modes. To each mode, there corresponds one telemetry layout. TM_FMT1 is the default mode entered after start-up. There are no mode change actions associated to the mode managers with the exception of the controller mode manager that, upon entry into RUN mode, loads a ramp profile manoeuvre.

A.1.2 Data Pools

The application has only one single data pool that contains the following items:

• sensor data
• actuator data
• controller error
• controller set-point

A.1.3 Failure Detection Policy

The failure detection is carried out through consistency checks and monitoring checks. The type of checks that are performed depends on the operational mode. In the SBY mode, the following checks are performed:

• Check SBY_1: the sensor output is subjected to an out-of-range check
• Check SBY_2: the sensor output is subjected to a delta check (i.e. data does not change value more than is a predefined delta threshold)

In the RUN mode, the following checks are performed:

• Check RUN_1: the sensor output is subjected to an out-of-range check
• Check RUN_2: the sensor output is subjected to a delta check
• Check RUN_3: the actuator output is subjected to an out-of-range check
• Check RUN_4: the control error is subjected to an out-of-range check
• Check RUN_5: the data pool is subjected to a consistency check

The recovery action associated to the above checks is always the same: system reset recovery action.
A.1.4 Failure Recovery Policy
The following recovery strategy is associated to the SBY mode:

- system reset on configuration error

The following recovery strategy is associated to the RUN mode:

- system reset on too many failure errors
- execution of local recovery actions

The two recovery actions are linked together where the system reset action on too many failure errors is executed first.

A.1.5 Telecommanding Policy
The application can process the following telecommands:

- GoToSBY: put the mission mode manager in the SBY mode
- GoToRUN: put the mission mode manager in RUN mode
- StartTriangularWaveProfile: start an angular wave profile manoeuvre
- StartRampProfile: start a ramp profile manoeuvre
- Reset: perform a software reset on the application

The remote operator sends a telecommand as a string that contains three or more integers. The interpretation of these integers is as follows:

- Integer 1: number of integers in the string
- Integer 2: telecommand identifier
- Integer 3: time tag
- Integers 4...n: telecommand specific data

The telecommand string is sent to the Jbed & PowerPC platform over the FTP socket. It is received by the telemetryReceiver thread that loads it into the telecommand loader. The telecommand loader decodes the string and dynamically instantiates a telecommand component to execute the telecommand. Note that the telecommand loader is activated in the non-real-time part of the application (it is activated by the telemetryReceiver thread) and therefore the poor timing predictability of the memory allocation operation does not pose problems. The memory associated to a telecommand object is automatically released by the Jbed garbage collector. Note that in the Jbed implementation of the JVM, operation of the garbage collection is compatible with real-time operation. After the telecommand loader has constructed a new telecommand objects, it loads it into the telecommand manager, which is then responsible for executing it.
A.1.6 Telemetry Policy

The application alternatively generates two telemetry formats, TM_FMT1 and TM_FMT2. The first telemetry format (TM_FMT1) contains a telemetry image of the following components:

- the Jbed clock component
- all event repositories

The second telemetry format TM_FMT2 contains a telemetry image of the following components:

- the Jbed clock component
- all event repositories
- the data pool

Telemetry data are sent by the telemetry manager to a telemetry stream component. In the case of the prototype application, the telemetry stream writes the telemetry data to an integer array, which acts as a telemetry buffer. From the implementation point of view, two telemetry buffers are maintained: the active and the idle buffer. The active buffer is the one which is being filled in by the telemetry manager. The idle buffer is available for forwarding to the remote operator. The telemetry stream toggles between these two buffers: when it has finished writing one frame to the currently active buffer, it makes the idle buffer into the active buffer and vice-versa.

The forwarding of the telemetry buffer data to the remote operator is done by a dedicated active component – the telemetry sender component, which is activated by a dedicated thread – the telemetryThread thread. When it is activated, the telemetry sender component retrieves the idle buffer from the telemetry stream component and processes it. Since the telemetry stream and the telemetry sender components should never be operating on the same telemetry buffer, the telemetry sender replaces the telemetry buffer it retrieves from the telemetry stream with a new and empty buffer. Note that the telemetry sender is activated by a non-real-time thread.

In summary, the telemetry buffers are being filled by a real-time task but are flushed by a non-real-time thread. This separation is necessary because the flushing of the telemetry data to the remote operator is not compatible with real-time constraints (it contains for instance dynamic memory allocation operations and the data are sent to the remote operator over a public network where delays are possible). The possibility exists that some telemetry frames can be lost, but the real-time task is protected from interference by the non-real-time thread. The telemetry buffer is assembled by the telemetry stream component as an array of integers with the following format:

- first element: the number of elements in the buffer
- second element: the frame number
- ...: the images of the telemeterable components in the currently active telemetry list

The telemetry buffer is sent by the telemetry sender component as a string with the following format: the string contains the elements of the telemetry buffer separated by blank characters and is terminated by a new line character.
A.1.7 Manoeuvres

Two manoeuvres are foreseen for the application:

- **RampProfile**: linearly ramp up the controller set-point from an initial value of zero to a final target value
- **TriangularWaveProfile**: make the controller set-point follow a triangular saw profile

The first manoeuvre is useful immediately after entry into RUN mode when the swing mass speed must be brought from zero to a target value. The second manoeuvre can be launched by telecommand at any time while in RUN mode.

A.1.8 Controllers

Only one controller is foreseen that is active in RUN mode and processes the sensor outputs and the controller set-point to generate an actuator command. The controller implements a PID control algorithm.

A.1.9 Sensors and Actuators

Only one sensor is present in the target control system:

- **RH_SPEED_SENSOR**: sensor to measure the rotating speed of the right-hand swing mass

Only one actuator is present in the target control system:

- **LH_SPEED_ACTUATOR**: actuator to control the speed of the left-hand motor

A.1.10 External Interfaces

The prototype application manages the following external interfaces:

- **A/D converter**: acquisition of sensor read-outs
- **D/A converter**: stimulation of the actuator
- **FTP Socket**: socket through which the telemetry buffer string is sent to the remote operator and the telecommand string is received from the remote operator.

A.2 The Control Application for the Swing Mass Model Deployed on the PERC Platform

A.2.1 Operational Modes

The following components in the prototype application are mode-dependent:
• failure detection manager
• failure recovery manager
• controller manager
• telemetry manager

The first three components have the following three modes:
• stand-by mode (SBY)
• speed mode (RUN_SPEED)
• position mode (RUN_POSITION)

Their modes are slaved to the mode of a mission mode manager component. This ensures that their mode changes are synchronized. The SBY mode is the default mode entered immediately after start-up. In SBY, the application processes telecommands and generates telemetry. There is no closed-loop control of the swing mass speed or position. In RUN_SPEED mode the right-hand disk’s speed is controlled and in RUN_POSITION mode its position is controlled.

The telemetry manager has the following two modes:
• telemetry format 1 (TM_FMT1)
• telemetry format 2 (TM_FMT2)

The telemetry manager cyclically alternates between these two modes. To each mode, there corresponds one telemetry layout. TM_FMT1 is the default mode entered after start-up.

There are no mode change actions associated to the mode managers with the exception of the system mode manager that, upon entry into the RUN_POSITION mode, sets the value of the reference equal to the value of the position sensor.

The secure switching between the modes e.g. from RUN_SPEED mode to RUN_POSITION mode is controlled by the remote operator.

A.2.2 Data Pools

The application has only one single data pool that contains the following items:
• speed sensor data
• position sensor data
• actuator data (data for the left-hand motor in the SMM)
• disturbance data (data for the right-hand motor in the SMM)
• controller error
• controller set-point
A.2.3 Failure Detection Policy

The failure detection is carried out through consistency checks and monitoring checks. The type of checks that are performed depends on the operational mode. In the SBY mode, the following checks are performed:

- Check SBY_1: the speed sensor output is subjected to an out-of-range check
- Check SBY_2: the speed sensor output is subjected to a delta check
- Check SBY_3: the position sensor output is subjected to a out-of-range check

In the RUN_SPEED mode, the following checks are performed:

- Check RUN_SPEED_1: the speed sensor output is subjected to an out-of-range check
- Check RUN_SPEED_2: the speed sensor output is subjected to a delta check
- Check RUN_SPEED_3: the position sensor output is subjected to a out-of-range check
- Check RUN_SPEED_4: the actuator output is subjected to an out-of-range check
- Check RUN_SPEED_5: the control error is subjected to an out-of-range check (disabled in code)
- Check RUN_SPEED_6: the data pool is subjected to a consistency check

In the RUN_POSITION mode, the following checks are performed:

- Check RUN_POSITION_1: the sensor output is subjected to an out-of-range check
- Check RUN_POSITION_2: the sensor output is subjected to a delta check
- Check RUN_POSITION_3: the position sensor output is subjected to a out-of-range check
- Check RUN_POSITION_4: the actuator output is subjected to an out-of-range check
- Check RUN_POSITION_5: the disturbance output is subjected to an out-of-range check
- Check RUN_POSITION_6: the control error is subjected to an out-of-range check (disabled in code)
- Check RUN_POSITION_7: the data pool is subjected to a consistency check

The recovery action associated to the above checks is always the same: system reset recovery action.

A.2.4 Failure Recovery Policy

The following recovery strategy is associated to the SBY mode:

- system reset on configuration error

The following recovery strategies are associated to the RUN modes:
• system reset on too many failure errors
• execution of local recovery actions

The two recovery actions are linked together and the first one is executed first.

A.2.5 Telecommanding Policy

The application can process the following telecommands:

• GoToRUN_SPEED: put the mission mode manager in run-speed mode
• GoToRUN_POSITION: put the mission mode manager in run-position mode
• StartTriangularWaveProfile: start an triangular wave profile manoeuvre
• StartRampProfile: start a ramp profile manoeuvre
• Reset: perform a software reset on the application
• SetControllerParameters: sets new controller PID parameters and optional the minimal disturbance output (only for RUN_POSITION mode available)

A telecommand is sent by the remote operator as a telecommand string that contains three or more integers. The interpretation of these integers is as follows:

• Integer 1: number of integers in the string
• Integer 2: telecommand identifier
• Integer 3: time tag
• Integers 4...n: telecommand specific data

The telecommand string is sent to the target PC over the FTP socket. It is received by the telecommandReceiver thread that loads it into the telecommand loader. The telecommand loader decodes the string and dynamically instantiates a telecommand component to execute the telecommand. Note that the telecommand loader is activated in the non-real-time part of the application (it is activated by the telecommandReceiver thread) and therefore the poor timing predictability of the memory allocation operation does not pose problems. The memory associated to a telecommand object is automatically released by the PVM (PERC VM) garbage collector. Note that in the PERC implementation of the JVM, operation of the garbage collection is compatible with real-time operation. After the telecommand loader has constructed a new telecommand objects, it loads it into the telecommand manager which is then responsible for executing it.

A.2.6 Telemetry Policy

The application alternatively generates two telemetry formats, TM_FMT1 and TM_FMT2. The first telemetry format (TM_FMT1) contains a telemetry image of the following components:
A.2. The Control Application for the Swing Mass Model Deployed on the PERC Platform

- the PERC clock component
- the mission mode manager component
- all event repositories

The second telemetry format (TM_FMT2) contains a telemetry image of the following components:

- the PERC clock component
- the mission mode manager component
- all event repositories
- the data pool

Telemetry data are sent by the telemetry manager to a telemetry stream component. The telemetry stream writes the telemetry data to an integer array which acts as a telemetry buffer. From an implementation point of view, two telemetry buffers are maintained: the active and the idle buffer. The active buffer is the one which is being filled in by the telemetry manager. The idle buffer is available for forwarding to the remote operator. The telemetry stream toggles between these two buffers: when it has finished writing one frame to the currently active buffer, it makes the idle buffer into the active buffer and vice-versa.

The mechanism of sending telemetry data is identical as for the control application for the SMM executed on the Jbed & PowerPC platform, for more details see Section A.1.6.

The telemetry buffer is assembled by the telemetry stream component as an array of integers with the following format:

- first element: the number of elements in the buffer
- second element: the frame number
- ...: the images of the telemeterable components in the currently active telemetry list

The telemetry buffer is sent by the telemetry sender component as a string with the following format: the string contains the elements of the telemetry buffer separated by blank characters and is terminated by a new line character.

A.2.7 Manoeuvres

Two manoeuvres are foreseen for the application:

- RampProfile: linearly ramp up the controller set-point from an initial value of zero to a final target value
- TriangularWaveProfile: make the controller set-point follow a triangular saw profile

The first manoeuvre is useful immediately after entry into one of the RUN modes when the swing mass speed or position must be brought to a target value. The second manoeuvre can be launched by telecommand at any time while being in one of the two RUN modes.
A.2.8 Controllers

Only one controller is foreseen that is active in every RUN mode and processes the sensor output and the controller set-point to generate an actuator command. Note that in the RUN_SPEED mode only the speed sensor output and in the RUN_POSITION mode only the position sensor output is processed. The controller implements a PID control algorithm. In the RUN_POSITION mode a slightly changed integrator part is used to take the static friction of the system into account. All used controller blocks – P, I and D blocks – are designed in Matlab Simulink and automatically translated to C code by using Real-Time Workshop. The code is then incorporated using the Matlab Wrapper classes into the application, for more information about the Matlab Wrapper see Sections 2.7.2 and 3.4.5.

A.2.9 Sensors and Actuators

Two sensors are present in the target control system:

- **RH_SPEED_SENSOR**: sensor to measure the rotating speed of the right-hand swing mass
- **RH_POSITION_SENSOR**: sensor to measure the angular position of the right-hand swing mass

Two actuators are present in the target control system:

- **LH_ACTUATOR**: actuator to control the torque of the left-hand motor
- **RH_ACTUATOR/DISTURBANCE**: actuator of the right-hand motor used as disturbance (only used in RUN_POSITION mode)

Neither the sensors nor the actuators are redundant.

A.2.10 External Interfaces

The prototype application manages the following external interfaces:

- **A/D converter**: acquisition of sensor read-outs (speed and position) by using the Me2600i PCI-Card
- **D/A converter**: stimulation of the actuator and disturbance by using the Me2600i PCI-Card
- **FTP Socket**: socket through which the telemetry buffer string is sent to the remote operator and the telecommand string is received from the remote operator.
A.3. The Control Application for the ER1 Robot Deployed on the Java Platform

A.3.1 Operational Modes

The following components in the prototype application are mode-dependent:

- controller manager
- telemetry manager

The controller manager component operates in the following two modes:

- stand-by mode (SBY)
- run mode (RUN)

The mode switching is operated by a mission mode manager component. The SBY mode is the default mode entered immediately after start-up. In the SBY mode, the application processes telecommands and generates telemetry. There is no closed-loop position control. In the RUN mode the application processes telecommands, generates telemetry, and the robot's position is controlled.

The telemetry manager has the following two modes:

- telemetry format 1 (TM_FMT1)
- telemetry format 2 (TM_FMT2)

The telemetry manager cyclically alternates between these two modes. To each mode, there corresponds one telemetry layout. TM_FMT1 is the default mode entered after start-up. There are no mode change actions associated to the mode managers.

A.3.2 Data Pools

The application has only one single data pool that contains the following X and Y position data:

- sensor data
- actuator data
- reference data
- control error data

A.3.3 Failure Detection Policy

The control application does not provide any failure detection policy.
A.3.4 Failure Recovery Policy

The control application does not implement any failure recovery policy.

A.3.5 Telecommanding Policy

The application can process the following telecommands:

- GoToSBY: put the mission mode manager in the SBY mode
- GoToRUN: put the mission mode manager in the RUN mode
- SystemReset: perform an application reset
- MoveToPositionManoeuvre: start a manoeuvre that defines the robot's target position

The remote operator sends as in case of the two previous applications a telecommand as a telecommand string that contains three or more integers. The interpretation of these integers is as follows:

- Integer 1: number of integers in the string
- Integer 2: telecommand identifier
- Integer 3: time tag
- Integers 4...n: telecommand specific data

The telecommand string is sent over the FTP socket. It is received by the telemetry-receiver thread that loads it into the telecommand loader. The telecommand loader decodes the string and dynamically instantiates a telecommand component to execute the telecommand. After the telecommand loader has constructed a new telecommand objects, it loads it into the telecommand manager, which is then responsible for executing it.

A.3.6 Telemetry Policy

The application alternatively generates two telemetry formats, TM_FMT1 and TM_FMT2. The first telemetry format (TM_FMT1) contains a telemetry image of the following components:

- the robot clock component
- the failure event repository
- the recovery event repository
- the system event repository
- the data pool
- the mission mode manager

The second telemetry format TM_FMT2 contains a telemetry image of the following components:
A.3. The Control Application for the ER1 Robot Deployed on the Java Platform

- the robot clock component
- the failure event repository
- the recovery event repository
- the system event repository
- the data pool

Telemetry data are sent by the telemetry manager to a telemetry stream component. The mechanism of toggling telemetry buffers (described in previous applications, see Section 7.6.1 and 7.6.2) is used to decouple real-time and non-real-time operations, and can be also applied to the robot control application that is deployed on a non-real-time platform.

It was reused also the format of the telemetry buffer:

- first element: the number of elements in the buffer
- second element: the frame number
- ...: the images of the telemeterable components in the currently active telemetry list

The telemetry buffer is sent by the telemetry sender component as a string with the following format: the string contains the elements of the telemetry buffer separated by blank characters and is terminated by a new line character.

### A.3.7 Manoeuvre

The control application operates only with one manoeuvre:

- **PositionManoeuvre**: set reference data for the X and Y position

The manoeuvre can be executed in both *SBY* and *RUN* modes at any time.

### A.3.8 Controllers

For each direction (X and Y) it is foreseen that only one P controller is active in the *RUN* mode. The P controller component is provided as a default framework component.

### A.3.9 Sensors and Actuators

The ER1 robot is a modular kit that can be easily re-configured to carry additional payload. However, not only robot’s design can be customized, but also the electronic can be expanded through use of digital and analog ports. Note that the core of the robot is a Robot Control Module (RCM) that interfaces the motors and I/O to the personal computer which runs special ER1 software.

Two stepper motors are used to actuate motor, with sensors reading X and Y position. The API/Command Line Interface allows to gather sensor data and send commands to actuators through FTP sockets. A direct access to manipulate robot’s sensors and actuators is unfortunately not supported.
A.3.10 External Interfaces

The robot control application manages following external interfaces:

- *FTP Socket 1*: socket through which the telemetry buffer string is sent to the remote operator and the telecommand string is received from the remote operator.
- *FTP Socket 2*: socket through which robot sensor data are received and actuator data are sent.
B.1 The Control Application for the Swing Mass Model Deployed on the Jbed Platform

B.1.1 Application Specific Factories

- **JbedSmmControllerFactory** is the application specific factory for controller management components. This factory creates a controller mode manager and configures the controller manager.

- **JbedSmmFailureDetectionFactory** is the factory for failure detection management components for the SMM application. This factory initializes the size of the consistency checkable and monitoring check lists and it offers a method to create and configure the mode manager.

- **JbedSmmFailureRecoveryFactory** offers methods to construct, configure and load the failure recovery mode manager and to construct, configure and load the recovery strategies for each operational mode of the failure recovery mode manager.

- **JbedSmmManoeuvreFactory** offers methods to create the concrete manoeuvres required for the SMM application. Only one copy of each manoeuvre is created.

- **JbedSmmModeFactory** does not implement any methods. It simply inherits all the methods from the base mode factory. It should be used to construct application specific mode change objects.
• **JbedSmmMonitoringFactory** does not implement any methods. It inherits all the methods from the base monitoring factory.

• **JbedSmmSystemFactory** creates the clock component and the data pool components.

• **JbedSmmTelecommandFactory** provides a method to create an application specific telecommand loader.

• **JbedSmmTelemetryFactory** initializes the size of the telemetrical lists and it offers a method to create and configure the mode manager, the telemetry stream and the telemetry sender component.

• **JbedSmmUnitFactory** provides a method to create the unit mode managers, analog sensor, and analog actuator.

• **TestSmmSystemFactory** is the factory for system management components of the SMM prototype application deployed on a desktop. It creates the clock component and the data pool components.

• **TestSmmUnitFactory** is the factory for unit management components of the SMM prototype application deployed on a desktop. This factory provides a method to create the unit mode managers, analog sensors, and analog actuators.

### B.1.2 Application Specific Components

• **JbedSmmDataPool** is a data pool for the Jbed the SMM application. The data pool stores and provides sensor data, actuator data, control error data, and reference data.

• **JbedSmmClassId** defines class identifiers for the SMM application classes.

• **JbedSmmRamp** is a manoeuvre component generating a ramp trajectory.

• **JbedSmmTriangularWave** is a manoeuvre component generating a wave trajectory.

• **JbedSmmClock** is a system time component for the SMM application deployed on the Jbed & PowerPC platform. The component offers a method for acquisition of the system time, clock period, and clock cycle.

• **TestSmmClock** is a system time component for the SMM application deployed on the desktop. The component offers a method for acquisition of the system time, clock period, and clock cycle.

• **JbedSmmPid** is a component implementing a PID controller.

• **JbedSmmTelecommandLoader** is a telecommand loader for the SMM application deployed on the Jbed & PowerPC platform. Four types of telecommands are recognized:

1. A telecommand to change the mode of the mission mode manager.

2. A telecommand to force a software reset.
3. A telecommand to load a manoeuvre to execute a triangular wave profile on the actuator set point.

4. A telecommand to load a manoeuvre to execute a ramp profile on the actuator set point.

- **JbedSmmTelemetryStream** implements the telemetry stream for the SMM application. The telemetry data are written to an integer array buffer. Boolean variables are encoded as usual with 1 being true and 0 being false. Floating point variables are multiplied by a scaling factor and then converted to integer. A saturation check is introduced to prevent overflow. The telemetry stream internally maintains two telemetry buffers: the active buffer and the idle buffer.

- **JbedSmmTmSender** is a component that is responsible for forwarding a telemetry buffer to a remote operator. This component performs operations that are not compatible with real-time constraints. It should therefore execute as a non-real-time thread. The telemetry buffer to be forwarded to the remote operator is acquired from the **JbedSmmTelemetryStream** component. The hardware channel through which telemetry buffer data are forwarded to the remote operator is represented by a standard PrintWriter component.

- **JbedSmm_1** is a component representing the control application for the SMM deployed on the Jbed & PowerPC platform.

- **TestSmm_1** is a test control application deployed on the desktop. This class is intended to be as close as possible to the **JbedSmm_1** class. It is used to test the control application prior to downloading to the target system.

- **JbedSmmAnalogActuator** represents an analog actuator component developed for the Jbed ADDA board (see Figure 7.6).

- **JbedSmmAnalogSensor** represents an analog sensor component developed for the Jbed ADDA board (see Figure 7.6).

- **TestSmmAnalogActuator** represents a test analog actuator component developed for the control application deployed on the desktop.

- **TestSmmAnalogSensor** represents a test analog sensor component developed for the control application deployed on the desktop.

- **JbedSmmConnectionInstaller** is a component that sets up the connection between a client and server (a remote operator and the control application).

A configuration file defining configuration constants of the application specific components is implemented by the **JbedSmmConfiguration** class that offers the configuration constants as class properties.
B.2 The Control Application for the Swing Mass Model Deployed on the PERC Platform

B.2.1 Application Specific Factories

Fourteen application factories were developed to assist to instantiate the control application for the SMM deployed on the PERC & VxWorks & PC platform:

- **PercSmmControllerFactory** is the application specific factory for controller management components. This factory creates a controller mode manager and configures the controller manager.

- **PercSmmFailureDetectionFactory** is the factory for failure detection management components for the SMM application. This factory initializes the size of the consistency checkable and monitoring check lists and it offers a method to create and configure the mode manager.

- **PercSmmFailureRecoveryFactory** offers methods to construct, configure and load the failure recovery mode manager and to construct, configure and load the recovery strategies for each operational mode of the failure recovery mode manager.

- **PercSmmManoeuvreFactory** offers methods to create the concrete manoeuvres required for the SMM application. Only one copy of each manoeuvre is created.

- **PercSmmModeFactory** offers a method to create a new mode change object of the type PercSmmModeChangeAction.

- **PercSmmMonitoringFactory** does not implement any methods. It inherits all the methods from the base monitoring factory.

- **PercSmmSystemFactory** creates the clock component and the data pool components.

- **PercSmmTelecommandFactory** provides a method to create an application specific telecommand loader.

- **PercSmmTelemetryFactory** initializes the size of the telemetrable lists and it offers a method to create and configure the mode manager, the telemetry stream and the telemetry sender component.

- **PercSmmUnitFactory** provides a method to create the unit mode managers, analog sensor, analog actuator, disturbance generators, and disturbance actuator.

- **TestSmmSystemFactory** is the factory for system management components of the SMM prototype application deployed on the desktop. It creates the clock component and the data pool components.

- **TestSmmUnitFactory** is the factory for unit management components of the SMM prototype application deployed on the desktop. This factory provides a method to create the unit mode managers, analog sensors, analog actuator, disturbance generators, and disturbance actuator.
B.2. The Control Application for the Swing Mass Model Deployed on the PERC Platform

- **TimePercSmmTelecommandFactory** is a factory for telecommand management components of the time measurement application. This factory provides a method to create an application specific telecommand loader.

- **TimePercSmmTelemetryFactory** is a factory for telemetry management components of the time measurement application. This factory initializes the size of the telemetry lists and it offers a method to create and configure the mode manager, the telemetry stream and the telemetry sender component.

B.2.2 Application Specific Components

- **PercSmmDataPool** is a data pool for the control application deployed on the PERC & VxWorks & PC platform. The data pool stores and provides sensor data, actuator data, control error data, and reference data.

- **PercSmmFollowerControllerModeManager** is an application specific encapsulation of a follower mode manager for the controller mode manager. This component defines a mode manager that is based on the follower mode manager. In a typical configuration, this mode manager would be slaved to the mission mode manager.

- **PercSmmClassId** defines class identifiers for the SMM application classes.

- **PercSmmRamp** is a manoeuvre component generating a ramp trajectory. This component has the same functionality as the JbedSmmRamp manoeuvre component developed for the Jbed & PowerPC platform.

- **PercSmmTriangularWave** is a manoeuvre component generating a wave trajectory. This component has the same functionality as the JbedSmmTriangularWave manoeuvre component developed for the Jbed & PowerPC platform.

- **PercSmmModeChangeAction** is a component providing a specific mode change action when the application enters into another mode.

- **PercSmmClock** is a system time component for the SMM application deployed on the PERC & VxWorks & PC platform. The component offers a method for acquisition of the system time, clock period, and clock cycle.

- **TestSmmClock** is a system time component for the SMM application deployed on the desktop. The component offers a method for acquisition of the system time, clock period, and clock cycle.

- **PercSmmD_Block** is a control block that implements a derivative action with a fixed gain.

- **PercSmmI_Block** is a control block that implements an integrator with a fixed gain and output limits.

- **PercSmmPIDCCSuperBlock** is an application specific ControlChannelSuperBlock component. This component allows setting of all controller parameters at runtime.
• **PercSmmControllerTelecommand** is an application specific telecommand enable to load new parameters into the controller.

• **PercSmmTelecommandLoader** is a telecommand loader for the SMM application deployed on the PERC & VxWorks & PC platform. Five types of telecommands are recognized:

  1. A telecommand to change the mode of the mission mode manager.
  2. A telecommand to force a software reset.
  3. A telecommand to load a manoeuvre to execute a triangular wave profile on the actuator set point.
  4. A telecommand to load a manoeuvre to execute a ramp profile on the actuator set point.
  5. A telecommand to set the controller parameters and optionally the minimal value of generated disturbance.

• **TimePercSmmTelecommandLoader** is a telecommand loader for the time measurement application. Five types of telecommands are recognized. They are identical as for the PercSmmTelecommandLoader component.

• **PercSmmTelemetryStream** implements the telemetry stream for the SMM application. The telemetry data are written to an integer array buffer. Boolean variables are encoded as usual with 1 being true and 0 being false. Floating point variables are multiplied by a scaling factor and then converted to integer. A saturation check is introduced to prevent overflow. The telemetry stream internally maintains two telemetry buffers: the active buffer and the idle buffer. This component offers the same functionality as the JbedSmmTelemetryStream component developed for the Jbed & PowerPC platform.

• **PercSmmTmSender** is a component that is responsible for forwarding a telemetry buffer to a remote operator. This component performs operations that are not compatible with real-time constraints. It should therefore executed as a non-real-time thread. The telemetry buffer to be forwarded to the remote operator is acquired from the PercSmmTelemetryStream component. The hardware channel through which telemetry buffer data are forwarded to the remote operator is represented by a standard PrintWriter component. This component offers the same functionality as the JbedSmmTmSender component developed for the Jbed & PowerPC platform.

• **TimePercSmmTmSender** is a component developed for the time measurement application. The component is responsible for forwarding a telemetry buffer to a remote operator, its functionality is identical with the PercSmmTmSender component.

• **PercSmm** is a component representing the control application for the SMM deployed on the PERC & VxWorks & PC platform.
B.3. The Control Application for the ER1 Robot Deployed on the Java Platform

- PercSmmTimer_1 is a component representing the control application for the SMM deployed on the PERC & VxWorks & PC platform. In comparison to the PercSmm_1 component, the PercSmmTimer_1 component uses Timer class to schedule application threads. The PercSmm_1 component schedules threads by calling of the Thread.sleep() method.

- TestSmm_1 is a test control application deployed on the desktop. This class is intended to be as close as possible to the PercSmm_1 class. It is used to test the control application prior to downloading to the target system.

- TimePercSmm_1 is the time measurement application. It measures the worst case execution times of the control application deployed on the Windows operating system.

- TimeVxPercSmm_1 is the time measurement application. It measures the worst case execution times of the control application deployed on the VxWorks real-time operating system.

- PercSmmAnalogActuator represents an analog actuator component developed for the PERC & VxWorks & PC platform using PCI ADDA board.

- PercSmmAnalogSensor represents an analog sensor component developed for the PERC & VxWorks & PC platform using PCI ADDA board.

- TestSmmAnalogActuator represents a test analog actuator component developed for the control application deployed on the desktop.

- TestSmmAnalogSensor represents a test analog sensor component developed for the control application deployed on the desktop.

- PercSmmDisturbanceGenerator is a disturbance generator component.

- PercSmmZeroDisturbanceGenerator is a disturbance generator component that is possible to switch off, i.e. its output will always be zero.

The configuration constants of the application specific components are stored in the PercSmmConfiguration class. They are implemented as class properties.

B.3 The Control Application for the ER1 Robot Deployed on the Java Platform

B.3.1 Application Specific Factories

- RobotControllerFactory is the application specific factory for controller management components. This factory creates a controller mode manager and configures the controller manager and initialize size of lists for storing controllers.

- RobotFailureDetectionFactory does not implement any methods. It simply inherits all the methods from the base mode factory.
• RobotFailureRecoveryFactory does not implement any methods. It simply inherits all the methods from the base mode factory.

• RobotManoeuvreFactory offers methods to create a X,Y position manoeuvre. Only one instance of the manoeuvre is created.

• RobotModeFactory does not implement any methods. It simply inherits all the methods from the base mode factory. It should be used to construct application specific mode change objects.

• RobotMonitoringFactory does not implement any methods. It inherits all the methods from the base monitoring factory.

• RobotSystemFactory creates the clock component and the data pool components.

• RobotTelecommandFactory provides a method to create an application specific telecommand loader.

• RobotTelemetryFactory initializes the size of the telemetrable lists and it offers a method to create and configure the mode manager, the telemetry stream and the telemetry sender component.

• RobotUnitFactory provides a method to create the unit mode managers, and different types of sensors and actuators. For example X and Y position sensor, distance and angle sensor, speed actuator, angle speed actuator, etc.

B.3.2 Application Specific Components

• RobotDataPool is a data pool for the robot application. The data pool stores and provides sensor data, actuator data, control error data, and reference data.

• RobotClassId defines class identifiers for the robot application classes.

• RobotXYPositionManoeuvre is a manoeuvre component generating a trajectory determined by cartesian coordinates.

• RobotClock is a system time component for the robot application deployed on the desktop. The component offers a method for acquisition of the system time, clock period, and clock cycle.

• RobotTelecommandLoader is a telecommand loader for the robot application deployed on the Java & PC platform. Three types of telecommands are recognized:

  1. A telecommand to change the mode of the mission mode manager.
  2. A telecommand to force a software reset.
  3. A telecommand to load a manoeuvre to generate a trajectory determined by cartesian coordinates on the actuator set point.
• **RobotTelemetrySender** is a component that is responsible for forwarding a telemetry buffer to a remote operator. This component performs operations that are not compatible with real-time constraints. It should be therefore executed as a non-real-time thread. The telemetry buffer to be forwarded to the remote operator is acquired from the **RobotTelemetryStream** component. The hardware channel through which telemetry buffer data are forwarded to the remote operator is represented by a standard PrintWriter component. This component offers the same functionality as the JbedSmmTmSender and PercSmmTmSender component developed for the Jbed & PowerPC and PERC & VxWorks & PC platform.

• **RobotTelemetryStream** implements the telemetry stream for the robot application. The telemetry data are written to an integer array buffer. Boolean variables are encoded as usual with 1 being true and 0 being false. Floating point variables are multiplied by a scaling factor and then converted to an integer. A saturation check is introduced to prevent overflow. The telemetry stream internally maintains two telemetry buffers: the active buffer and the idle buffer. This component offers the same functionality as the JbedSmmTelemetryStream and PercSmmTelemetryStream component developed for the Jbed & PowerPC and PERC & VxWorks & PC platform.

• **Robot** is a component representing the control application for the robot deployed on the Java & PC platform.

• **RobotXYPositionSensor** is a sensor component measuring robot’s X and Y position.

• **RobotXPositionActuator** is an actuator component performing movement in the X direction.

• **RobotYPositionActuator** is an actuator component performing movement in the Y direction.

• **RobotExternalConnectionInstaller** is a component that sets up the connection between a remote operator and the robot control application.

• **RobotInternalConnectionInstaller** is a component that sets up the connection between a robot control application and an application accessing hardware of the robot.

The component configuration constants are implemented by the **RobotConfiguration** class. They are encoded as class properties.


Curriculum Vitae

Václav Čechtický

Born July 30th, 1974 in Prague, Czechoslovakia

1980 - 1988 Primary School, Prague

1988 - 1992 High School, Prague

1992 Matura

1993-1997 Bachelor study at the Czech Technical University, Department of Electrical Engineering.

1997 Bachelor Diploma in Electrical Engineering (Bc. CTU)

1997-1999 Undergraduate student at the Czech Technical University, Department of Electrical Engineering.

1999 Diploma in Electrical Engineering (Dipl. El.-Ing. CTU)


2000 - 2004 Research and Teaching Assistant with Prof. Dr. W. Schaufelberger of the Automatic Control Laboratory, Swiss Federal Institute of Technology.