Towards Use-Case Driven Self-Management of Distributed Systems

Reza Haydarlou
Towards Use-Case Driven Self-Management of Distributed Systems

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus Prof. ir. K.Ch.A.M. Luyben,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen op 29 november 2011 om 15:00 uur
door

Ali Reza HAYDARLOU

doctorandus in de informatica
geboren te Khoy, Iran.
Dit proefschrift is goedgekeurd door de promotor:

Prof. dr. F.M.T. Brazier

Samenstelling promotiecommissie:

Rector Magnificus voorzitter
Prof. dr. F.M.T. Brazier Technische Universiteit Delft, promotor
Dr. M.A. Oey Technische Universiteit Delft, co-promotor
Prof. dr. S. Dobson University of St Andrews
Prof. dr. A. Plaat Universiteit van Tilburg
Prof. dr. A. van Deursen Technische Universiteit Delft
Prof. dr. Y-H. Tan Technische Universiteit Delft
Dr. P.H.G. van Langen Technische Universiteit Delft, advisor

Cover design by Shadi Haydarlou
Copyright © 2011 by A.R. Haydarlou

All rights reserved. No part of the material protected by this copyright notice may be reproduced or utilised in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage and retrieval system, without the prior permission of the author.

ISBN 978-90-8570-625-0
To my late parents who always encouraged me to explore science, and to my children, Shadi and Poeya, hoping they will pursue science...
Acknowledgements

For the realisation of this thesis, I have benefited from the support, guidance, and encouragement of many people. They helped me in completing my PhD research journey. I am deeply indebted to them. First of all, I would like to thank my promotor Frances Brazier. She gave me the opportunity to enter into the fascinating world of scientific research. She guided me to elevate the different aspects of the research topics to a higher level of abstraction. I am also grateful to Benno Overeinder who was my weekly supervisor at the time the IIDS group was established at the VU University Amsterdam.

I owe special thanks to two people, Michel Oey (my co-promotor) and Martijn Warnier. I learned a lot from both of you during the writing of this thesis. You inspired me, contributed to my thinking, and helped me to advance my knowledge in different areas. You spent a lot of time and effort to review each chapter page by page, and gave me both high level and in-depth feedback.

My gratitude also goes to all members of my thesis committee for their effort to read my thesis and provide me with valuable comments. Your comments have deepened my thinking on some topics in this thesis.

During my PhD research trajectory at the VU University Amsterdam and Delft University of Technology, I had the pleasure to work with wonderful colleagues. I would especially like to thank Sander van Splunter, Kassidy Clark, and Evangelos Pournaras for supporting and encouraging me in writing this thesis. I would also like to thank Elth Ogston, David Mobach, Thomas Quillian, Reinier Timmer, Rick van Krevelen, Jordan Janeiro, Jan-Paul van Staaldruinen, Çağrı Tekinay, Mohsen Davarynejad, Tanja Buttler, Jonatan Bijl, Maartje van den Bogaard, Yilin Huang, Thiene Hennis, Selin Ebeci, Mingxin Zhang, Alireza Rezaee, and other colleagues of the Systems Engineering section; thank you all for creating a warm, friendly, interesting, funny, and great atmosphere. Also many thanks to Sabrina Rodrigues and Everdine de Vreede-Volkers for their valuable administrative support.

I would like to express my gratitude to my family and friends, especially to my dear wife Karin Dekker. Without her patience, continuing support, and love, this thesis would not have been accomplished. Thank you for supporting and encouraging me through all these years.

Delft, Reza Haydarlou
October 2011
Acknowledgements
Contents

Acknowledgements v

1 Introduction 1
   1.1 Management of Distributed systems .......................... 2
   1.2 Management of Behaviour .................................. 2
   1.3 Autonomic Computing Model ................................ 3
   1.4 Research Objective ........................................ 5
   1.5 Research Question ......................................... 6
   1.6 Research Approach ......................................... 6
   1.7 Contributions .............................................. 7
   1.8 Thesis Outline ............................................. 8

2 Related Work & Positioning 11
   2.1 Autonomic Systems: System Organisation ...................... 11
      2.1.1 Unstructured Autonomic Systems ...................... 12
      2.1.2 Dynamically Structured Autonomic Systems ............ 14
      2.1.3 Statically Structured Autonomic Systems ............. 15
   2.2 Autonomic Systems: Domains of Application .................. 17
      2.2.1 Management Unit Perspective .......................... 17
      2.2.2 Knowledge Utilisation Perspective .................... 19
   2.3 Framework Positioning ..................................... 19

3 Use-case Driven Self-Management 23
   3.1 Introduction .............................................. 24
      3.1.1 Behavioural Complexity ................................ 24
      3.1.2 Structural Complexity ................................ 25
   3.2 Basic Unit of Management .................................. 26
   3.3 Reusing Available Knowledge ................................ 28
   3.4 Use-Cases .................................................. 29
   3.5 Use-Case Levels .......................................... 31
   3.6 Use-Case References ...................................... 34
      3.6.1 Horizontal References .................................. 34
      3.6.2 Vertical References .................................... 35
   3.7 An Example Scenario ........................................ 37
3.7.1 System Level View ................................................. 39
3.7.2 Runnable Level View ............................................. 39
3.7.3 Component Level View ............................................ 42
3.7.4 Class Level View .................................................. 43
3.8 Summary ............................................................... 44

4 The Management Model ................................................ 45
4.1 Introduction .......................................................... 46
4.2 Management Model Overview ........................................ 46
  4.2.1 High Level Overview ............................................ 46
  4.2.2 A Failure Scenario ............................................... 48
4.3 Analyser ............................................................... 49
4.4 Diagnoser ............................................................. 51
4.5 Planner ............................................................... 56
4.6 Plan Translator ....................................................... 58
4.7 Autonomic Manager .................................................. 59
  4.7.1 Autonomic Manager’s Process .................................... 60
  4.7.2 Relationship between Autonomic Managers .................... 62
4.8 Information Flow Entities ............................................. 64
  4.8.1 Sensors ........................................................... 64
  4.8.2 Symptoms ........................................................ 67
  4.8.3 Hypotheses ....................................................... 68
  4.8.4 Plans .............................................................. 69
  4.8.5 Effectors ........................................................ 72
4.9 Related Work ........................................................ 72
4.10 Summary ............................................................. 80

5 The System Model ....................................................... 81
5.1 Behavioural Model .................................................... 81
  5.1.1 Job ............................................................... 82
  5.1.2 Task .............................................................. 83
  5.1.3 State ............................................................. 85
  5.1.4 Event ............................................................ 86
5.2 Structural Model ...................................................... 87
  5.2.1 Managed System .................................................. 88
  5.2.2 Managed Runnable ............................................... 88
  5.2.3 Managed Connector .............................................. 90
  5.2.4 Managed Component ............................................. 91
  5.2.5 Managed Class .................................................. 92
  5.2.6 Managed Method ............................................... 92
5.3 Summary ............................................................. 93
### 6 Self-Management Knowledge Representation

6.1 Knowledge Representation Requirements .......................... 95
   6.1.1 Knowledge Locality & Modularity .......................... 95
   6.1.2 Knowledge Reasoning ........................................ 96
   6.1.3 Knowledge Acquisition ....................................... 96

6.2 Choice for Knowledge Representation .............................. 96

6.3 Semantic Web Overview ........................................... 97
   6.3.1 OWL Concepts & Properties .................................. 98
   6.3.2 OWL Cardinality Restrictions ............................... 99
   6.3.3 OWL Value & Existential Restrictions ..................... 99
   6.3.4 OWL Consistency Check ...................................... 99
   6.3.5 Requirements Satisfaction ................................... 99

6.4 Self-Management Ontology ......................................... 100
   6.4.1 Autonomic-Manager Sub-Ontology ............................ 101
   6.4.2 Behavioural-Model Sub-Ontology ............................. 101
   6.4.3 Structural-Model Sub-Ontology ............................... 102
   6.4.4 Analyser Sub-Ontology ....................................... 102
   6.4.5 Diagnoser Sub-Ontology ...................................... 103
   6.4.6 Sensor Sub-Ontology .......................................... 104

6.5 Related Work .................................................... 106

6.6 Summary .......................................................... 107

### 7 Illustrative Scenarios

7.1 Methodology ...................................................... 109

7.2 Case 1: Single-Level Use-Case Management ....................... 111
   7.2.1 Managed System Description ................................ 111
   7.2.2 Use-Case Descriptions ....................................... 111
   7.2.3 Self-Management Model Specification ....................... 114
   7.2.4 Simultaneous Failure Diagnosis ............................. 118

7.3 Case 2: Multi-Level Use-Case Management ......................... 121
   7.3.1 Managed System Description ................................ 122
   7.3.2 Use-Case Descriptions ....................................... 122
   7.3.3 Self-Management Model Specification ....................... 125
   7.3.4 Multi-Level Failure Diagnosis .............................. 130

7.4 Concluding Remarks ............................................... 131

7.5 Summary .......................................................... 132

### 8 The Execution Environment

8.1 Execution Environment Overview .................................. 135

8.2 Activation & Control Engine ..................................... 136
   8.2.1 Communication between Autonomic Management & Managed System .................................. 137
   8.2.2 Synchronisation between Autonomic Management & Managed System ............................. 138
   8.2.3 Communication between Autonomic Managers .................. 138
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.2.4 Invocation of Inspective Plan</td>
<td>139</td>
</tr>
<tr>
<td>8.2.5 Bootstrapping Autonomic Management</td>
<td>139</td>
</tr>
<tr>
<td>8.3 Rule Engine</td>
<td>139</td>
</tr>
<tr>
<td>8.4 Instrumented Managed System</td>
<td>140</td>
</tr>
<tr>
<td>8.4.1 Instrumentation Technique</td>
<td>141</td>
</tr>
<tr>
<td>8.4.2 Instrumentation Code</td>
<td>142</td>
</tr>
<tr>
<td>8.5 Summary</td>
<td>143</td>
</tr>
<tr>
<td>9 Conclusions and Future Work</td>
<td>145</td>
</tr>
<tr>
<td>9.1 Thesis Summary</td>
<td>145</td>
</tr>
<tr>
<td>9.2 Research Question Revisited</td>
<td>147</td>
</tr>
<tr>
<td>9.3 Discussion</td>
<td>148</td>
</tr>
<tr>
<td>9.4 Future Work</td>
<td>150</td>
</tr>
<tr>
<td>A Generic Rules</td>
<td>153</td>
</tr>
<tr>
<td>A.1 Symptom Occurrence Rules Template</td>
<td>153</td>
</tr>
<tr>
<td>A.2 Hypothesis Selection Rules</td>
<td>155</td>
</tr>
<tr>
<td>A.3 Hypothesis Validation Rules</td>
<td>156</td>
</tr>
<tr>
<td>A.4 Hypothesis Evaluation Rules</td>
<td>157</td>
</tr>
<tr>
<td>A.5 Diagnosis Determination Rules</td>
<td>158</td>
</tr>
<tr>
<td>A.6 Plan Selection Rules</td>
<td>158</td>
</tr>
<tr>
<td>A.7 Plan Translation Rules</td>
<td>159</td>
</tr>
<tr>
<td>B Autonomic Management Code</td>
<td>161</td>
</tr>
<tr>
<td>B.1 Performing Autonomic Process</td>
<td>161</td>
</tr>
<tr>
<td>B.2 Instantiating Autonomic Manager</td>
<td>162</td>
</tr>
<tr>
<td>B.3 Performing Diagnostic Process</td>
<td>163</td>
</tr>
<tr>
<td>B.4 Handling Sensor Values</td>
<td>164</td>
</tr>
<tr>
<td>B.5 Execution of Rules by Rule Engine</td>
<td>165</td>
</tr>
<tr>
<td>Samenvatting (Dutch Summary)</td>
<td>167</td>
</tr>
<tr>
<td>Bibliography</td>
<td>171</td>
</tr>
<tr>
<td>Index</td>
<td>189</td>
</tr>
<tr>
<td>Curriculum Vitae</td>
<td>191</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Management of today’s complex information systems is an important challenge. An information system is usually defined as a set of hardware, software, network, people, and procedures that are configured to collect and process data into information [102, 154, 171]. Almost all commercial and social organisations depend on information technology for their continual survival. Information is considered as a vital resource produced by their information systems [158]. As the dependency of organisations on information grows day by day, requirements as information quality, processing, delivery, and distribution become more and more stringent and diverse.

Within most organisations, information needs to be in the right place, at the right time, and in the right form. Information must often need to be consistent with other information in the system. Information needs to be concise (no extra information) and to represent external facts accurately. Timely and reliable information delivery and service availability is critically important.

Information systems are expected to quickly react to rapidly changing organisational, political, economic, social, and technological situations. They are expected to be flexible and handle large volume of information. Information systems are also expected to exchange information within different departments of an organisation, between organisations, and with customers. These systems often span different departments and organisations at various physical and geographical locations, requiring distributed information processing. Complex distributed information systems are the result.

In summary, the demands for high quality information, timely and distributed information delivery and information access, and cross-organisation information integration increase the complexity of distributed information systems. The complexity of these systems must be managed timely, efficiently, and cost effectively. The focus of this thesis is on the management of distributed information systems.
1.1 Management of Distributed systems

A distributed system is usually defined as a system in which computing elements (program units), located at networked nodes, communicate by message passing [57, 177] to cooperatively solve some complex problem [162]. For example, the different program units of a distributed multimedia system, located on multimedia databases, proxy and information servers, and the various mobile clients, work together to provide multimedia content to mobile users [124]. Another example is that in a distributed financial trading system, several program units located on transaction servers, applications servers, web servers, groupware servers, and workstations cooperate to perform financial transactions. Significant characteristics of distributed systems are concurrency, scalability, and resource sharing. As a result, the coordination of concurrently executing computing elements sharing resources becomes an important topic in design and implementation of distributed systems.

The growing complexity of distributed systems is a crucial constraint on the success in designing and management of these systems. The need to provide increasing degrees of dynamism, heterogeneity, decentralisation, and decoupling between distributed program units increases the complexity of distributed systems [32]. More specifically, the complexity is increased because the program units are usually written in different programming languages, commonly interact with each other through several different interaction mechanisms and protocols, are often executed both inside and outside organisations, and utilise different architectural models.

Complexity of distributed systems can be described as a measure of the difficulty of understanding and managing these systems. The complexity is related to the size of a system, the degree of differentiation of a system’s computing elements, and the degree of chaotic behaviour a system has [155]. The size of a system is measured by the number of nodes and services, and the number of interconnections or dependencies between its compartments. The chaotic behaviour is the effect of a small variation in a certain part of a system on the overall system behaviour. The complexity of distributed systems influences their management.

The goal of the management of distributed systems is to ensure the effective and efficient operation of these systems in line with organisational objectives. The management tasks and activities depend on a specific management purpose. Examples of management purposes are configuration management, security management, availability management, performance management, and recovery management.

1.2 Management of Behaviour

The complexity of distributed systems does not only concern their structure. The need to design distributed systems for automating sophisticated governmental, scientific, and corporate business processes, including business to business (B2B) electronic commerce, increases the complexity of behaviour of distributed systems. System behaviour is defined as a set of actions or reactions of a system in response
to external or internal stimuli [128]. Examples of system behaviour are handling a money transaction by an Internet banking system and showing articles in response to a search query by a web shopping system. Usually, sophisticated system behaviour is divided into a number of simpler behaviours that are executed by different distributed computing elements. Showing articles by a web shopping system is an example of a complex behaviour. This behaviour consists of analysing a given search query, retrieving requested data from a database, and converting the data into a human readable format. Management of behaviour of distributed systems is the primary focus of this thesis.

The responsibility of management of the behaviour of a distributed system is to ensure that the system correctly exposes expected and specified behaviour. Manual management of behaviour of such systems requires more and more human skills. Determining the root cause of software runtime failures and performance degradations in such complex systems is difficult, costly, and very time consuming [56, 72, 91, 109, 181]. Automating routine management tasks relieves system administrators from the burden of manually detecting and resolving system malfunctions.

Ideally distributed systems would be able to recognise and solve a large portion of their malfunctions on their own. To this purpose, these systems would need to know when and where an incorrect state occurs, and have adequate knowledge to analyse the problem situation, diagnose the root-cause, and make healing and optimisation plans. They should also be able to execute these plans to stabilise, heal, or optimise themselves, if possible, without human intervention. The autonomic computing [109] initiative started by IBM in 2001 incorporates these generic management functionalities in one model.

Autonomic computing has been proposed to reduce the cost of maintaining complex systems by developing computer systems capable of self-management. Ganek et al. [72] distinguish a number of autonomic properties, such as self-configuring, self-healing, self-optimising, and self-protecting. Extending and enhancing a system with these properties is an important step towards a self-managed system. This thesis utilises the general principles of the autonomic computing to make the self-management of behaviours of existing distributed systems possible. The following section explains the general principles of autonomic computing.

1.3 Autonomic Computing Model

Many researchers have proposed (and extensively studied) the use of control theory techniques for system management purposes [53, 86, 149]. Control theory describes the behaviour of dynamic systems (such as mechanical, electrical, or biological) and provides a basic mechanism with which a dynamic system can maintain its own steady state [122]. A very common mechanism provided by control theory is the closed-loop feedback.

Figure 1.1 shows the concept of closed-loop feedback in which the actual output of a system state is controlled based on the previous actual output and a desired output value. It is a continuous feedback loop for process flow. Whenever there is a
deviation from a desired value of a system state, the controller takes corrective actions. A controller receives signals from sensors placed in the device, computes the difference between the desired and actual values, and performs corrective actions based on this computation.

The autonomic computing initiative [93] has greatly benefited from the formalisms provided by control theory. The very abstract reference architecture for autonomic computing [72], shown in Figure 1.2, is based on the closed-loop feedback control. The underlying principle is surprisingly simple. The decision entity receives measurements from a resource entity, makes decisions and sends adaptive instructions to the resource entity. The adaptive instructions often influence the state of the resource entity, that in turn sends new measurements to the decision entity.

The blueprint architecture for autonomic computing [93] refines this abstract reference architecture, identifying a number of fundamental concepts and architectural building blocks for self-managed systems. A self-managed system has two main building blocks: an autonomic manager and a managed resource, as depicted in Figure 1.3.

According to the blueprint, a managed resource is a hardware or software component that corresponds to the resource entity in the control loop. Examples of managed hardware resources are workstations, routers, and storage devices. Examples of managed software resources are operating systems, database servers, web servers, application servers, and business applications. A managed resource has a manageability interface that includes sensors and effectors. An autonomic manager uses this manageability interface to obtain management data, gathered by sensors, and to change the behaviour of the managed resource through effectors.

The autonomic manager is the decision entity in the control loop. The autonomic manager’s control loop distinguishes four main entities, the MAPE entities: monitor, analyse, plan, and execute. The responsibilities of these entities are re-
1.4 Research Objective

The ultimate goal of this thesis is to explore the potentials of autonomic computing for management of behaviour of existing distributed systems. The architectural concepts, presented in the autonomic computing architectural blueprint [93], are not sufficient to achieve this goal. These concepts define a common approach and terminology, but do not specify a particular implementation. Consequently, there is a need for more specific self-management concepts that concern the key principles of distributed systems. One of the key principles of distributed systems is that they comprise from a number of heterogeneous sub-systems communicating with each other through network using various protocols. As a result, most often, heterogeneous sub-systems are delivered by different vendors, and maintained by different departments in an enterprise. In general, knowledge regarding each sub-system has been spread out over different domain experts with their own view from system, and usually the quality of system maintenance documentation is not very high. Therefore, communication between domain experts maintaining a system is less efficient, and manual management suffers from delayed problem determination. Self-management can be used to improve problem determination. However, domain experts would probably not be willing to cooperate to implement self-management of existing systems if it requires much effort. In conclusion, a
successful self-management approach should:

- be able to efficiently *capture* system knowledge from domain experts and *integrate* their views in one unified model,
- be able to *detect*, *diagnose*, and *repair* the most frequently occurring system malfunctions by utilising domain experts’ knowledge,
- be less *intrusive* on existing systems and domain experts.

### 1.5 Research Question

Following from the problem description (complexity of behaviour of distributed systems and their management), the issues regarding the management of distributed systems, and the requirements regarding a successful self-management approach, this thesis investigates the central research question:

**RQ: How to design autonomic management of behaviour of existing distributed systems?**

To answer the central question, certain aspects of the artifact to be managed (distributed system) and its management need to be understood, analysed, and taken into account. With respect to the distributed systems, this entails:

- **identifying different types of behaviours to be managed,**
- **determining the relationship between behaviours,**
- **specifying knowledge about behaviours and their relationships.**

With respect to the management of distributed systems, this entails:

- **managing an individual behaviour,**
- **coordinating the management of multiple behaviours,**
- **specifying knowledge required for the management of behaviours.**

The above research question is used to guide the design of the *system model* and the *management model*, presented in this thesis. Furthermore, the two models are combined into a single model called *self-management model*.

### 1.6 Research Approach

The *research philosophy* [46] followed by this thesis is a combination of *positivism* and *interpretivism*. Positivists believe that scientific knowledge is built up from verifiable observations and inferences drawn from controlled experiments [88]. They study complex phenomena by decomposing them into simpler components.
Interpretivists attempt to understand how people think about the world by collecting qualitative data about people’s activities [61]. In this thesis, both the complex behaviour and the structure of a distributed system are decomposed into simpler and manageable units. Also, qualitative data regarding operational management processes are collected from system administrators, functional analysts, and system developers.

The research strategy followed by this thesis is design science. The design science strategy guides a scientific research to provide a solution for a specific problem with emphasis on artifacts as the outcome of the research [87, 156]. Artifacts can be concepts, models, methods, or implemented/prototyped systems. The resulting products of this thesis are system and management related concepts and models, and an implemented framework.

The research instruments used by this thesis are literature review, case study, action research, and evaluation. A literature review provides background knowledge and deeper understanding of the research domains [165]. A case study provides a qualitative and observatory insight into the real-life context of the research phenomenon [192]. Action research encourages researchers to experiment through intervention and incorporate the effects of their intervention into their theories [15]. Evaluation helps researchers assess the final product of their research. This thesis employs the literature review and case study instruments to perform an extensive literature survey regarding autonomic systems and study the case of a distributed trading system. It also utilizes the action research instrument to study the effects of specific alterations in the relationships between autonomic management and managed system. The evaluation instrument is used to apply the implemented framework to a number of real-life cases.

1.7 Contributions

This thesis advances the state of the art in self-management of distributed systems by introducing the first general purpose self-management model:

- that is based on behavioural aspects of distributed systems through the use of use-cases,
- that is declarative,
- that is a bridge between the high-level functional world and the low-level technical world by allowing the integration of knowledge from both worlds in one unified model,
- that can be used for the self-management of existing distributed systems.

The work on the research question leads to several more specific contributions that are summarised as follows:
A distinction is made between behaviours at system, operational, functional, and code levels based on the software fault handling process, currently encountered in many enterprises. All of these behaviours are represented in use-case notations. A use-case is considered to be the unit of management. Use-cases at each level describe the interactions with the system from the viewpoint of a designated domain expert, such as a system administrator, functional analyst, software developer, etc. As a result, a use-case based autonomic manager views the system from a domain expert's perspective.

Each of the use-cases, independent of the level at which it resides, may reference other use-cases. Two types of use-case references are identified: horizontal and vertical references. The former is a reference from a use-case at a specific level to a use-case at the same level, and the latter is a reference to a use-case at a different level.

A Semantic Web based model of distributed systems is proposed to formally describe the behaviour and structure of these systems.

An autonomic manager, including its main entities (analyser, diagnoser, planner and plan translator), for managing the execution of a use-case is designed. The information such as symptom, hypothesis, and plan, exchanged between these entities, are specified as well.

Corresponding to the referential relationships (both horizontal and vertical) among use-cases, the autonomic managers managing these use-cases are related with each other as well. A mechanism is proposed to realise the communication and cooperation between autonomic managers to manage the overall system behaviours properly.

A Semantic Web based management model of distributed systems is proposed to formally describe each entity of the autonomic manager, information flow between these entities, and the relationship between the autonomic managers.

A self-management framework based on the system and management models is designed and implemented. The framework can automatically instrument sensors and effectors in an existing distributed system, and generate code for autonomic management purposes, based on declarative behaviour descriptions. The framework focuses on self-diagnosis.

1.8 Thesis Outline

The structure of the remainder of this thesis is depicted in Figure 1.4. Chapter 2 gives an overview of the research works related to autonomic systems. Chapter 3 explains why a use-case driven self-management approach is preferable. Chapters 4 and 5 together introduce the self-management model. Chapter 4 presents the management model and describes how an autonomic manager manages a use-case
of a system, and how multiple autonomic managers of a system cooperate with each other to manage the behaviour of the whole system. Chapter 5 lays out the structural and behavioural model of a distributed system used by autonomic managers.

Chapter 6 discusses requirements for representing self-management knowledge in distributed environments, and argues that the Semantic Web languages OWL and SWRL satisfy the requirements. Chapters 7 and 8 present the application of the framework and the execution environment. Chapter 7 describes the results of two case studies to illustrate how the framework can be applied to real-life applications. Chapter 8 describes how the framework can be used to provide autonomic management for a specific distributed system based on the models specified in OWL and SWRL. Finally, Chapter 9 presents the conclusions and a discussion on future work.
Chapter 2

Related Work & Positioning

This chapter provides an overview of research on autonomic systems related to this thesis. Autonomic systems proposed in the related works are studied from two perspectives: system organisation and domains of application. The first perspective, system organisation, focuses on the relationship between autonomic elements in an autonomic system. An autonomic element is the fundamental building block of any autonomic system, and contains a control loop that integrates an autonomic manager with its managed resource (see Figure 1.3). The second perspective, domains of application, concentrates on the area for which an autonomic system has been designed and developed. The domains of application are reviewed on the basis of two important characteristics of autonomic systems: management unit and knowledge utilisation. The first characteristic concerns the managed resource of each autonomic manager in an autonomic system. The second characteristic involves the way autonomic elements utilise/exchange knowledge about a managed system.

2.1 Autonomic Systems: System Organisation

Autonomic systems can be characterised by the way their autonomic elements are organised, that is, the way autonomic elements are coordinated and the way they exchange information. The internal organisation of autonomic systems is also affected by the way in which they fulfill specific self-* properties. The organisation of autonomic elements in autonomic systems can vary from less structured to more structured as depicted in the spectrum shown in Figure 2.1. On the left side of the spectrum are emergent autonomic systems within which autonomic elements are fully decentralised and interact with each other in changing configurations (Unstructured Autonomic Systems). In the center of the spectrum are self-organising autonomic systems in which the relationships between autonomic elements arise/change during the life-time of a system (Dynamically Structured Autonomic Systems). On the right side of the spectrum are autonomic systems in which the hierarchical relationships between autonomic elements are pre-defined.
Related Work & Positioning

and static (Statically Structured Autonomic Systems). In the following sections, examples of unstructured, dynamically structured, and statically structured autonomic systems are reviewed, focussing in particular on the interaction between autonomic elements.

Figure 2.1: The spectrum of the system organisation of autonomic systems ranging from less structured to more structured.

2.1.1 Unstructured Autonomic Systems

In unstructured autonomic systems, system-wide autonomic behaviour is the result of local interactions between individual autonomic elements that solely follow their own local rules and are not explicitly aware of the global impact of these rules [189]. In such fully decentralised systems, there is no central entity to control the behaviour of individual autonomic elements. This property is the essence and strength of a fully decentralised and emergent system [6].

In unstructured autonomic systems, the role of coordination mechanisms is of key importance. A short description of the most used coordination mechanisms, from pheromone-based through market-based to token-based, can be found in [52, 189]. Because of the broad similarities between multi-agent systems and fully decentralised autonomic systems, Brazier et al. [25] propose to apply communication protocols and coordination mechanisms utilised by multi-agent systems to autonomic systems. The common assumption is that autonomic elements communicate asynchronously, take independent actions based on incomplete local knowledge, and influence only their own peers without any direct feedback. This assumption holds, independent of whether autonomic elements are natural entities modifying their local environment by laying down pheromone trails (pheromone-based), economic entities buying or selling computing resources by using electronic currencies (market-based), or individual entities holding and passing around tokens containing a localised view of the system (token-based).

In fully decentralised environments, market-based coordination mechanisms are usually used to optimise resource allocation or task assignment between self-interested and competing autonomic elements [3, 78, 79, 195]. The global objective of such a system is to prevent self-interested autonomic elements from wasting and monopolising scarce resources, and to minimise the time required to complete
computational tasks. Note that such systems do not have a central auctioneer. Local interactions of buyers and sellers obey the laws of supply and demand. The auctioneer is one of the bidders. Zimmerman et al. [195], for example, apply a market-based mechanism to wireless sensor networking. They propose one utility (profit) function for buyers and one for sellers. The utility function for buyers calculates the amount of time that can be saved by adding an additional processing node based on the computational task on which they are working. The utility function for sellers calculates the expected profit from each proposed bid and chooses the bid that generates the greatest profit. Ajourlou et al. [3] use the concepts of price, revenue, cost, and one-round auctions in a decentralised environment in which a team of Unmanned Aerial Vehicles (UAVs) collects data from a set of Unmanned Ground Sensors (UGSs) and delivers it to a Ground Station (GS). UGSs are deployed in a remote area far from the GSs. Ajourlou et al. consider visiting a UGS by a UAV, collecting data, and returning to the GS to be a continuous task, which they refer to as a sensor-visit task. A number of sensor-visit tasks are broadcast to the UAVs. The price of a task is the amount of virtual money that will be given to a UAV for completing the task within a given time window. A UAV then submits a bid on the task most profitable for it. The GS evaluates all bids received and chooses the one with the most profit. Ajourlou et al. show that the system-wide objective of optimising data delivery time can be achieved on the basis of local interactions between auctioneers.

One of the major challenges in the management of unstructured autonomic systems is the frequent changes of the number and type of the system's components. Wireless network systems [117, 160], automated guided vehicle (AGV) transportation systems [189, 190], swarm robotics systems [136], and traffic control systems [135, 188] are examples of such structurally flexible systems. To obtain maximum flexibility and scalability, and to handle frequent changes, fully decentralised autonomic approaches have been proposed. In these approaches, each base station in a wireless network, a car equipped with special gear transmitting wireless radio signals and containing a global positioning device in a traffic control system, and each AGV in a warehouse transportation system is an autonomic element that locally and fully autonomously exchanges and synchronises information with its neighbouring base stations, cars, or AGVs.

The way information is processed by autonomic elements, and the way information is exchanged between neighbouring autonomic elements are determined by the self-* properties for which an autonomic system has been designed. For example, Parunak et al. [147] describe self-optimisation in a wireless network system. The self-optimisation is realised by minimising power consumption of individual base stations and, at the same time, maximising system throughput. To achieve this self-optimisation, autonomic elements (base stations) build a history of their interactions with clients, utilising a reinforcement learning algorithm to give more weight to recent interactions, and to share this local history with their direct neighbours.

Another example is self-optimisation performed in an AGV transportation system [190]. Self-optimisation is realised by guaranteeing constant throughput of
packets in a warehouse. In this system, multiple AGVs pick up incoming packets, move them through AGV stations connecting segments, and drop them off at their destinations. Note that packets arrive at random times, AGVs can fail, AGVs can encounter obstacles, etc. To guarantee constant throughput, the system needs to ensure even distribution of AGVs between two zones: the zone to where the AGVs move to pick-up locations without a packet, and the zone to where the AGVs move to drop-off locations holding a packet. To achieve this goal, autonomic elements propagate artificial pheromones (that gradually disappear as they become older) to the neighbouring AGV stations. An artificial pheromone includes information about its own age, distance and cost of traveling from the current AVG station to the location of the packet, and the priority of the packet. So, AGVs on each AVG station can decide about the best outgoing segment (i.e., the direction of the closest packet and with the highest priority) based on a pheromone.

The major benefits of unstructured autonomic systems are the design simplicity of individual autonomic elements, low communication overhead, and scalability. Design simplicity is implied from the fact that autonomic elements are required only to know about their own activities. Based on their local knowledge regarding these activities, autonomic elements send informational (not instructional) messages to their neighbours at random intervals. As a result, the cost of communication is low, and autonomic elements can dynamically join or leave the system. However, it is often difficult to test and configure these systems because of the randomness: random message loss, random interaction between random pairs of autonomic elements, and random arrival time of events [6]. Unstructured autonomic systems are often used for modeling natural or artificial systems that are inherently distributed and fully decentralised.

2.1.2 Dynamically Structured Autonomic Systems

A dynamically structured autonomic system is a self-organising system in which the organisation of autonomic elements changes during runtime. These systems restructure themselves in response to changing environmental conditions to achieve a specific self-* property. As a result, the relationship between autonomic elements arises dynamically. The arrangement (topology) of autonomic elements in these systems differs from star to tree topologies. Such systems are used to self-optimise, self-configure, and self-heal network nodes in a wide range of applications ranging from radio access networks such as Universal Mobile Telecommunications System (UMTS) or Long Term Evolution (LTE) [97], to self-categorisation of visual objects in computer vision [71, 114], to provision of services in a Service Component Architecture (SCA) platform [144].

Dynamically structured autonomic systems use diverse mechanisms to restructure themselves. For instance, to search multi-media datasets in a large distributed p2p environment, Sedmidubsky et al. [161] base the interconnection of autonomic elements on the query-answer paradigm. Each peer (autonomic element) initially sends a query with MPEG-7 feature descriptors to peers from a random list. Those
peers that provide the highest *quality of answer* are added to the list of communicating peers. This list changes continuously, i.e., the system restructures itself according to answers returned to queries.

Visual object categorisation requires recognition of one or several objects in an image. Due to the wide variations of shape and appearance of objects inside a category, various scales and orientations of objects, and diverse illumination and occlusion of objects in an image, visual object categorisation remains a challenging task [71]. Kinnunen et al. [114] use an unsupervised competitive learning approach using the Kohonen *Self-Organising Map* (SOM) [115] algorithm to this purpose. SOMs provide a way to represent multi-dimensional data in much lower dimensional spaces. SOMs learn to classify data without supervision. Each node in the network has a specific topological position and contains a vector of weights of the same dimension as the input vectors. A SOM learns to classify the training data by automatically adjusting the weights of nodes and re-organising them. For visual object categorisation, first, interest points at images are detected. Then, these points are converted to scale and rotation invariant descriptors. Finally, a codebook is constructed from these key point descriptions using SOM.

Boesen et al. [22] propose a biology inspired cell architecture (eCell) in which electronic DNA (eDNA) enables self-organisation. Similar to a biological cell, each eCell reads the eDNA and determines which function to perform. The eCells communicate with one another in a mesh topology. Each eCell, based on its interpretation of the eDNA, and on inputs from other eCells or from the environment, determines to which eCells to communicate. Any changes of the eDNA or unavailability of an eCell cause system re-organisation.

A more generic approach regarding self-organisation in a distributed p2p environment is developed by Pournaras et al. [152]. They introduce a generic middleware service (AETOS) that provides self-organised tree overlays for applications based on their requirements such as robustness, node degree, or expected response time. The nodes of a tree overlay are selected from a proximity list containing nodes with close proximity neighbours. In turn, the nodes in the proximity list are selected from a list of nodes constructed through random sampling. Self-organisation of the tree overlay is realised by the fact that both random and proximity lists are continually updated through a gossiping protocol [101].

The above examples illustrate the major benefits of dynamically structured autonomic systems: adaptability and flexibility. Note however, too much adaptability and flexibility can result in chaos and less robust systems [58]. Especially, if small environmental changes have strong effects, acquiring stable system-wide behaviour is a challenge.

### 2.1.3 Statically Structured Autonomic Systems

A statically structured autonomic system is composed of *hierarchically organised* autonomic elements each of which controls and optimises its own dedicated part of a system. The system hierarchy can contain many levels. An autonomic element
at a higher level (parent), in addition to its own responsibility of managing part of the system, is also responsible for managing a number of autonomic elements at the next lower level (children). Only parent-children interactions take place. Note that the root autonomic element in a statically structured autonomic system, in contrast to centralised systems¹, does not have knowledge over all constituent system components or autonomic elements.

An example of a hierarchically organised environment is a data center. A data center houses computer systems, storage systems, network devices, and their associated environmental controls such as power supplies, air conditioning, etc. Kephart, Tesauro, and Lefurgy [108, 121, 179] tackle the problem of dynamic resource allocation under changing workload conditions in a data center. They all model a data center as logically separated Application Environments (AE), each containing a pool of resources of various types, such as application servers, web servers, database servers, etc. AEs are hierarchically organised. Detailed control of resources within an AE, at each level, is handled by an autonomic manager. Autonomic managers at different levels communicate with each other by passing the result of their utility functions². The goal of the root autonomic manager is to optimise the overall utility function for the entire data center.

Khargharia et al. [112, 113] tackle the problem of energy consumption in a data center. They define autonomic managers at every physical hierarchy of a data center (processor, memory, IO, server, server cluster, and the whole data center). Each managed resource is modelled as a Discrete Event System (DEVS) which is a finite state machine, and includes a set of inputs, set of outputs, set of states, and a set of state transitions. The important task of each autonomic manager is to enable transitions from one state to another without violating any performance constraint while optimising the managed resource’s power consumption. In this approach, the direction of communication between autonomic managers is top-down. In other words, an autonomic manager at a higher level passes its decision (the output of its DEVS) to its children. This decision becomes the input of the DEVS of each child.

Another example of a hierarchically organised environment is a Federated Database Management System (FDBMS). An FDBMS is a collection of heterogeneous databases that provides a homogeneous view to distributed data managed by partly autonomous database systems [182]. The need to include complex data types in SQL statements and to store very large objects in databases has led to unmanageable complexity [74, 76, 126, 129, 131]. Automated query optimisation and dynamic workload management in an FDBMS are two active research areas that tackle the problem of increasing performance degradation. For this purpose,

¹To our knowledge, the current literature has not reported on fully centralised autonomic systems. In a fully centralised autonomic system, a central autonomic element has complete knowledge over all constituent autonomic elements. Therefore, this type is not covered in this chapter.

²A utility function maps a possible state of an autonomic element into a real scalar value (usually expressed in monetary units) or into a vector. A state is any combination of attributes, such as system availability, reliability, performance, and robustness, measured by sensors monitoring the different resources of an autonomic element.
2.2 Autonomic Systems: Domains of Application

various researchers [66, 74, 76, 129, 131, 191] propose to use a hierarchical autonomic system in which each database in an FDBMS is managed by a dedicated autonomic manager. For query optimisation, they use a statistical cost model to calculate the cost of all or a subset of query execution plans and select the cheapest one. For workload management, they use clustering and supervised learning techniques to discover homogeneous classes of database queries which have similar workload properties (e.g., execution time, memory utilisation, or average rate of random I/O regarding a query). The cost and workload models are continuously adjusted based on monitoring information exchanged between autonomic managers.

Note that in a statically structured autonomic system, both the number of levels and the number of autonomic elements in each level are known in advance. This is both an advantage and a limitation of these systems. These systems do not have to deal with the complex task of maintaining a continuously changing structure. However, they are not flexible; it is not possible to add a new autonomic element in the system during runtime. These systems are often used for modeling inherently hierarchically organised computing environments where structure rarely changes.

2.2 Autonomic Systems: Domains of Application

The review of autonomic systems in the previous section shows the diversity of application domains of autonomic systems. At a very generic level, the application domain is hardware management or software management. At a slightly more specific level, autonomic systems are applied to the domains of wireless network, logistics vehicle routing, traffic control, p2p overlay network, multi-media search on Internet, data center, federated database management, and e-commerce, among others. The similarities and differences between the application domains can be adequately described on the basis of two important characteristics of autonomic systems: management unit and knowledge utilisation (see Figure 2.2).

2.2.1 Management Unit Perspective

The autonomic computing architectural blueprint [93] organises each autonomic element into two building blocks: an autonomic manager and a managed resource. In general, the managed resource (management unit) can be a part of system structure (structural element) or system behaviour (behavioural element). A behavioural element can be macroscopic (the behaviour of the system as a whole) or microscopic (the behaviour of an individual autonomic element) [189]. A system can have multiple macroscopic and microscopic behaviours.

In the first years of research on autonomic systems, reviewed in the previous section, focus is mainly on hardware management. Management units in these systems are physical elements (structural elements) such as base stations in a wireless network system [117, 160], transport vehicles in a logistics vehicle system [189, 190], traffic lights or cars equipped with special gear in a traffic control system [118, 188,
The domains of application of autonomic systems are reviewed from the management unit and knowledge utilisation perspectives.

In the last decade, autonomic systems have also been designed for software management. Management units in the majority of software-focused autonomic systems are software elements (structural elements) such as multimedia search engines in a P2P multimedia application [161] or database servers in a federated database management [74, 76, 131, 191]. In the majority of generic software frameworks for autonomic computing, management units are structural elements such as web services in an autonomic service-oriented framework [10, 11, 14], components in an autonomic component-oriented framework [125], software agents in an autonomic multi-agent framework [20], or architectural elements in an autonomic architectural description framework [39, 116, 187]. Microscopic behaviour of a component or macroscopic behaviour of the whole system does not seem to be of central interest for these researchers.

Some researchers [13, 33, 34, 35, 59, 103, 159] do pay attention to the behaviour of a distributed system. For instance, Ballagny et al. [13] introduce a state-based component model in which each software component a state machine is employed. The correct behaviour of a component is modelled in the associated state machine in the form of a sequence of state transitions. Any deviation from this sequence is considered to be an abnormal behaviour. More or less the same approach can be found in [63] that mainly focuses on the microscopic behaviour of a component, and does not handle the management of the system’s macroscopic behaviours. In contrast, other researchers (e.g., [59, 103, 123]) focus only on a system’s macroscopic behaviour. They model this behaviour as a failure-trace or request-trace. A failure-trace contains information about abnormal behaviour logged by several components of a system during a period of time. A request-trace contains information about the paths that user requests follow as they move through the system. Note that these approaches do not model the relationship be-
2.3 Framework Positioning

The goal of this thesis is to provide a self-management framework that can be used to enrich an existing distributed system with self-management capabilities. The previous sections review autonomic systems from the management unit, knowledge utilisation, and system organisation perspectives. In this section, the proposed self-management framework is briefly explained from the same perspectives. The first part illustrates the management unit of the autonomic element in the framework, the second focuses on the way self-management knowledge is utilised.
Table 2.1: Summary of the review of application domains of autonomic systems from management unit and knowledge utilisation perspectives.

<table>
<thead>
<tr>
<th>Application Domain</th>
<th>Management Unit</th>
<th>Knowledge Utilisation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Hardware Management</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wireless network</td>
<td>- Base station</td>
<td>- Reinforcement learning</td>
</tr>
<tr>
<td>Wireless sensor network</td>
<td>- Unmanned Ground Sensor (UGS)</td>
<td>- Market-based profit function</td>
</tr>
<tr>
<td></td>
<td>- Unmanned Aerial Vehicle (UAV)</td>
<td></td>
</tr>
<tr>
<td>Logistics vehicle routing</td>
<td>- Automated guided vehicle (AGV)</td>
<td>- Digital pheromone</td>
</tr>
<tr>
<td>Traffic management</td>
<td>- Car equipped with special gear</td>
<td>- Local knowledge exchange</td>
</tr>
<tr>
<td>Traffic light control</td>
<td>- Traffic light</td>
<td>- Fuzzy logic</td>
</tr>
<tr>
<td>Dynamic resource allocation in data center</td>
<td>- Application Environment (AE)</td>
<td>- Utility function</td>
</tr>
<tr>
<td>Energy consumption in data center</td>
<td>- Physical elements (server, processor, etc.)</td>
<td>- Discrete event function</td>
</tr>
<tr>
<td><strong>Software Management</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Federated database management</td>
<td>- Database server</td>
<td>- Statistical cost model</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Supervised learning</td>
</tr>
<tr>
<td>Visual object categorisation</td>
<td>- Node of Self-Organising Map (SOM)</td>
<td>- Unsupervised learning</td>
</tr>
<tr>
<td>Multi-media search</td>
<td>- Search engine</td>
<td>- Query-answer paradigm</td>
</tr>
<tr>
<td>E-commerce</td>
<td>- Macroscopic behaviour (request/failure trace)</td>
<td>- Hierarchical clustering</td>
</tr>
<tr>
<td></td>
<td>- Microscopic behaviour (component state)</td>
<td>- Decision tree</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Automaton model</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- State transition model</td>
</tr>
<tr>
<td>Web service composition &amp; diagnosis</td>
<td>- Web service</td>
<td>- Clustering</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Supervised learning</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Ontology</td>
</tr>
<tr>
<td>Forest fire analysis based on com-</td>
<td>- Software component</td>
<td>- Rules specified in XML</td>
</tr>
<tr>
<td>ponent-oriented framework (ACCORD)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>System administration based on mul-</td>
<td>- Software Agent</td>
<td>- Learning methods</td>
</tr>
<tr>
<td>ti-agent framework (ABLE)</td>
<td></td>
<td>- Fuzzy logic</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Logical rules</td>
</tr>
<tr>
<td>Video conferencing based on archi-</td>
<td>- Architectural element</td>
<td>- Annotations specified in an Architectural Description Language (ADL)</td>
</tr>
<tr>
<td>tectural description framework (RAIN-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BOW)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The type of management unit in an autonomic approach is significant. Current research focusses primarily on structural elements of distributed systems rather than on their behavioural elements. In most cases, a network of autonomic managers are made responsible for managing the different structural elements of a distributed system (e.g., base stations in a wireless network) instead of managing the different behaviours of the system. Research that does focus on behaviour of distributed systems does not make a distinction between macroscopic and micro-
scopic behaviours, and does not explicitly model relationships between these types of behaviours [13, 33, 63, 103]. This thesis advocates that both macroscopic and microscopic behaviour of distributed systems need to be addressed, and considered as the management unit. Note that the choice for system behaviour as the unit of management does not exclude the system’s structural elements from the management scope of the autonomic manager, because a system behaviour is executed by a group of structural elements. As a result, this group becomes part of a collection of resources to be managed by an autonomic manager.

As stated above, knowledge about a managed system is of special importance. Self-management requires systems to know when and where abnormal behaviour occurs, to analyse the problem situation, to make healing plans, and suggest various solutions to the system administrator or heal themselves. Implicit self-management knowledge for software management purposes does not seem to be appropriate. For instance, using machine learning and statistical methods to analyse a failure or request trace raises a number of issues. First, appropriate feature selection for learning methods and suitable learning and statistical model construction regarding the structure and behaviour of software systems are not straightforward. Second, as it is inefficient (and in some cases impossible) to include all normal traces in the training sample set, it is difficult to estimate the generalisation capacity of a limited number of normal traces (training samples). High generalisation implies high false negative and low generalisation implies high false positive [35, 103].

This thesis argues that self-management knowledge needs to be made explicit by representing it in a formal language. Such a formal knowledge can be understood and shared by different autonomic elements of an autonomic distributed system. As there is no widely accepted method of automatically constructing this formal knowledge, it needs to be provided by system developers. This thesis uses Semantic Web languages to represent the formal knowledge. As system developers utilise use-cases to describe the behaviour of these systems, a use-case based approach is taken to apply the principles of autonomic computing to the self-management of distributed systems. Both macroscopic and microscopic behaviours are represented as use-cases, and explicitly related to each other. A use-case is the unit of management in this approach. The execution of each use-case is managed by a dedicated autonomic manager.

Use-cases and their corresponding autonomic managers are organised in a hierarchy as a statically structured decentralised autonomic system. Each autonomic manager in the hierarchy, on the one hand, is responsible for management of a system behaviour in its own scope. On the other hand, it influences the management decision of its parents (i.e., making multi-level management possible) by reporting its autonomic management result to its parents.

The framework is based on three models: management model, structural system model, and behavioural system model. The proposed models will be extensively explained in the coming chapters. These models make it possible to generate self-management code for existing distributed systems from declarative knowledge about both structure and behaviour of a managed system.
Chapter 3

Use-case Driven Self-Management

The complexity and size of information systems, especially distributed systems, are increasing significantly both with behaviour and structure. As a result, management and maintenance of such systems are becoming a serious challenge. Autonomic computing [109] is one of the approaches currently being developed to address this challenge by automating management and maintenance tasks. One of the objectives of autonomic computing is self-management. This thesis focuses on self-management of existing distributed systems through application of the general principles of autonomic computing, described in Chapter 1. Self-management must deal with both behavioural and structural complexities.

The autonomic computing architectural blueprint, depicted in Figure 1, distinguishes two building blocks in each and every autonomic system: an autonomic manager and a managed resource. To apply the general principles of autonomic computing to the management of distributed systems (self-management), the unit of management (i.e., managed resource) must be identified. Section 3.2 argues that self-management of distributed systems can be best achieved by choosing a behavioural element as the unit of management instead of a structural element, as is most common in the literature.

Self-management of distributed systems, however, requires knowledge about behaviour of these systems. Much of the knowledge needed is usually elaborated during the software development phase. Section 3.3 argues that this knowledge can be reused for automated management of distributed systems. The knowledge regarding the behaviour is commonly specified in the form of use-cases. Section 3.4 describes the notion of use-cases and their basic characteristics. Self-management of distributed systems also requires knowledge about structure of these systems. As the current use-case definition does not include knowledge regarding the structure of distributed systems, Section 3.4 introduces a new definition of a use-case from a management perspective. The new definition integrates behavioural and structural knowledge, providing the knowledge needed for self-management.

Section 3.5 introduces hierarchical use-case levels (i.e., behaviour levels) which organise use-cases at distinct levels. The use-case levels are based on the software fault handling process, currently encountered in many enterprises. Use-cases at
each level describe the interactions with the system from the viewpoint of a designated domain expert, such as a system administrator, functional analyst, etc. As a result, a use-case based autonomic manager also views the system from a domain expert’s perspective. The proposed organisation of use-cases yields distributed multi-level diagnosis. This thesis primarily focusses on diagnosis as the most important prerequisite for the main autonomic properties (self-configuration, self-healing, self-optimisation, and self-protection). Finally, Section 3.7 illustrates the use of a hierarchy of use-cases for an example scenario to determine the root-cause of a system malfunctioning at different levels.

3.1 Introduction

The main objective of self-management of a distributed system is to let the system itself manage the correct, effective, and efficient execution of its own behaviours. For this purpose, many researchers (e.g., [37, 39, 145]) propose to manage the system’s structural elements executing the system’s behaviours. This thesis argues that self-management of distributed systems could alternatively focus primarily on the behavioural elements of a system (i.e., its services).

As a software system’s behaviours are often described as use-cases, the self-management approach presented in this thesis is based on use-cases. Use-cases are explained in detail in Section 3.4, for now it is sufficient to know that use-cases provide a semi-formal notation to describe how a system should be used and the behaviour it provides. The following sections describe the complexity of distributed systems from both a behavioural and a structural perspective.

3.1.1 Behavioural Complexity

Attempts to automate highly complex corporate business processes\footnote{Note that the terms business (sub)process and operating system process are not the same.} including business to business (B2B) electronic commerce interactions are currently common [102, 171, 132]. As multiple complex business functionalities are combined and offered in one large corporate-wide distributed system, management is becoming even more complex. To cope with these complexities in a system capable of self-management, a system has to be able to obtain knowledge on its internal behaviour at any given moment, that is, current activities, the flow of control from one activity to another, the outcome of multiple activities and their synchronisation, etc. Transaction systems are an example of a complex distributed system. An example of a transaction system in the financial sector is a Trading System\footnote{The Trading System example is reused in the remainder of this thesis to illustrate the approach proposed in this thesis.}, that provides authenticated end-users the possibility to trade, perform payments, maintain customers’ shares, and estimate risks. This thesis uses the trade payment process, depicted in Figure 3.1, to illustrate complex system behaviour. Figure 3.1 depicts the main activities involved.
3.1 Introduction

Figure 3.1: The activity diagram of a trade payment process. Distributed systems normally contain a considerable number of such processes that cause the behavioural complexities to increase.

The process starts when a trade arrives. The information contained in the trade is validated. After ensuring that the trade is valid, two activities (getting the customer profile and getting the current financial market information) are activated to estimate the risk. If the outcome of the risk analysis is below a certain threshold then the customer’s bank account information is retrieved and the customer’s bank is contacted to effectuate payment. After successful payment, the position (i.e., the number of shares) of the customer is updated. Note that most activities shown in the figure, including B2B processes such as getting the current financial market information and realising payment, are in turn separate business processes. The trade payment process gives an impression of how complex contemporary business processes are.

3.1.2 Structural Complexity

Structural complexity of distributed systems concerns the infra-structure upon which they execute. Many enterprises run their distributed systems on a large number of heterogeneous machines often with different operating systems and characteristics. The infra-structure is dynamic: machines and the software running on them join or leave the infra-structure continuously. To provide more flexibility, each piece of the distributed system running on the mentioned infra-structure has a considerable number of configuration parameters.
Figure 3.2: An infra-structure, encountered in many enterprises, upon which distributed systems execute. It gives an impression of the structural complexities of distributed systems.

Figure 3.2 shows an example of an infra-structure implemented in many enterprises. At the right, end-users behind desktops interact with the enterprise’s systems behind the firewalls. An authentication server is placed between two firewalls in the Demilitarised Zone (DMZ). All other servers, such as web server, groupware server, application server, database server, broker server, file server, market data server, message queue server, and business process server are positioned behind the second firewall.

The trade payment process, as described above in Section 3.1.1, uses (1) the application server to receive and validate the trade, (2) the database server to retrieve the customer’s profile, (3) the market data server to get current market data, (4) the business process server to estimate risk, and (5) the groupware server to contact the customer’s bank and realise the payment. Each of these activities is implemented as a sub-process running on a specific server. The individual servers are structural elements.

To cope with the structural complexities, a system needs to possess information about each server and their communications, that is, which address is associated with each server, which servers communicate with each other, which protocols they use, in which order the servers should be started, etc.

3.2 Basic Unit of Management

To face the complexities concerning management of existing distributed systems, this thesis utilises the principles of autonomic computing. These principles dictate

---

3 The word server is an overloaded term. Here, a server does not refer to a machine (a hardware device) but it refers to a specialised software program running on a machine.
3.2 Basic Unit of Management

that distributed systems need to be able to monitor themselves, analyse situations, diagnose problems, and take proper remedial actions [109]. Applying the autonomic computing paradigm requires identification of a managed resource (see Figure 1.3) from a management point of view. There are at least two options. A distributed system can be viewed as a collection of:

- structural elements (servers), on each of which runs a sub-process (an activity) of a business (or non-business) process. This view is called a structural perspective.
- behavioural elements (business or non-business process), each of which is effectuated by a number of structural elements. This view is called a behavioural perspective.

The unit of management (the managed resource) for an autonomic manager managing an existing distributed system would therefore be either a structural or a behavioural element, as depicted in Figure 3.3.

![Figure 3.3](image)

**Figure 3.3:** Two options for managed resource for an autonomic manager: (a) A structural element (such as a system’s sub-system), or (b) a behavioural element.

Almost all autonomic properties such as self-configuring, self-healing, self-optimising, and self-protecting require diagnosis. Diagnosis is needed for identification of root-causes of system misconfigurations, system malfunctions, system degradations, and system security leaks. The following depicts the effect of the choice for the structural or the behavioural perspective on the autonomic manager’s diagnostic process regarding system malfunctions.

Generally speaking, a root-cause is the initiating cause of a causal chain. The primary goal of root-cause diagnosis is to understand why and how malfunctioning has occurred so that it can be prevented from reoccurring. The diagnostic process determines which part of system caused a system malfunctioning, and under which conditions (during the execution of which activity) the malfunctioning occurred. Without proper diagnosis, an autonomic manager is not able to construct and perform appropriate remedy plans. The autonomic manager obtains information regarding the system malfunctioning during system execution. The obtained information needs to be analysed and interpreted in the light of a certain context. Many researchers have stressed the importance of contextual knowledge.
Use-case Driven Self-Management

in performing diagnostic tasks ([143, 183]). The contextual knowledge refers to the information about the environment of the problem to be diagnosed.

The availability of appropriate contextual knowledge for an autonomic manager depends on the perspective (structural or behavioural) for which an autonomic system has been designed. A distributed system has multiple structural elements \(s_1..s_m\) and multiple behavioural elements \(b_1..b_n\). Depending on the choice for a structural or behavioural element as the unit of management, a self-managed system includes either autonomic managers \(A_{s_1}..A_{s_m}\) associated with the structural elements \(s_1..s_m\), or autonomic managers \(A_{b_1}..A_{b_n}\) associated with the behavioural elements \(b_1..b_n\). Each behaviour \(b_i\) contains a number of activities \(a_{i1}..a_{ik}\) each of which is executed by exactly one structural element and each \(s_i\) executes multiple activities of different behaviours. In fact, each of \(A_{s_1}..A_{s_m}\) manages the realisation of the activities belonging to different behaviours, but each of \(A_{b_1}..A_{b_n}\) manages the realisation of the activities belonging to one behaviour \(b_i\).

If a system has been designed from a behavioural perspective, an autonomic manager is aware of all activities of its managed behavioural element. In case of malfunctioning of an activity of that behavioural element, the autonomic manager knows how the activity has been invoked. The contextual knowledge in this case consists of a chain of behavioural invocations. If a system has been designed from a structural perspective, activities of different behaviours are executed within the same structural element. An autonomic manager of that structural element cannot know how that activity has been invoked (i.e., it lacks the contextual knowledge).

Note that the choice for one of the elements (structural or behavioural element) as the unit of management does not exclude the other from the operating scope of the autonomic manager. This thesis prefers the behavioural perspective, and advocates utilising use-cases to reuse existing knowledge about the business (or non-business) processes.

3.3 Reusing Available Knowledge

The autonomic manager requires access to knowledge about the managed unit. For example, monitoring a managed resource requires information from sensors that have been placed (instrumented) in the managed resource. Where sensors should be instrumented and which information they should provide to an autonomic manager depends on the availability and quality of knowledge about a managed unit. Self-management of distributed systems can reuse the knowledge regarding a system’s behaviour and structure acquired during software design and development. A large number of software engineers use Unified Modeling Language (UML) and Object Constraint Language (OCL) [157] to express knowledge on systems business processes (behavioural elements) and their software architecture (relation between structural elements). UML is a graphical language for visualising and specifying the artifacts of distributed systems. It defines various types of diagrams that can be used to clarify a system’s behaviour and structure. The UML use-case diagram, activity diagram, sequence diagram, and state diagram describe a system’s behaviour, and the UML class diagram and component diagram describe
3.4 Use-Cases

a system’s structure. OCL is a formal language designed to describe expressions and constraints in UML diagrams. Both languages are used by software engineers to understand, document, and describe different aspects of complex business processes and their structure.

Use-cases are core to UML behavioural diagrams. Use-cases are used to capture and describe requirements regarding complex business processes. Use-cases support communication between business users and technical domain experts. They model business processes by clearly specifying the boundaries and goals of a business process, and identifying the user interactions with the system. For example, the diagram depicted in Figure 3.1 is the UML activity diagram for the trade payment process. The use-case notion is most often used to describe business processes (functional requirements) but it can also be used to describe non-business processes (non-functional requirements) such as security, performance, availability, etc [41].

UML structural diagrams are used to show the static structure of a system, focusing on a system’s structural elements (e.g., servers), their relationships, and the type of relationships (communication protocols) irrespective of time. These diagrams allow software designers, architects, and developers to communicate about the high level organisation of the software and test/verify the soundness of the design. The relation between structural elements is represented by interfaces consisting of one or more methods (functions).

The knowledge of a system’s business process and its structure, specified in use-cases, UML diagrams, and OCL constraints is most often used for manual software maintenance purposes. This thesis assumes that this knowledge can also be used for automated management of software systems.

Coming back to the definitions of the structural and behavioural perspectives presented earlier, both the knowledge concerning a system’s structure and the knowledge concerning a system’s behaviour are incomplete on their own. The structural elements in the UML structural diagrams are black boxes without any detail on how a structural element executes a system’s behaviour. Likewise, use-cases do not include information about the structural elements by which they are executed. As the main objective of a self-management system is to manage the system’s behaviour, this thesis proposes to base self-management on use-cases and to extend them to also include the information regarding structural elements.

Use-cases are the driving concept behind the proposed self-management model (described in Chapters 4 and 5). The coming sections describe what use-cases are, how the basic characteristics of a use-case are extended, how use-cases can be organised, and how different use-cases are related.

3.4 Use-Cases

Jacobson introduced use-cases in 1992 [95] for requirements modeling and structuring. Software engineers utilise business use-cases to describe the behaviour of a system from the viewpoint of a user.
A use-case represents a unit of functionality that a user expects from a system. In spite of the fact that there is no single universally accepted definition of a use-case in the literature, there is a general acceptance [41, 69, 95] that a use-case is a collection of possible sequences of interactions between the system and external actors having a set of main goals to be reached with the help of the system.

Figure 3.4(a) shows the Trading System surrounded by a number of use-cases. The use-cases include User Authentication (user authenticates him/her-self to the system), Trade Entry (user enters a trade in the system), Payment (user requests the system to administrate the payment regarding a trade), and Update Shares (user requests the system to update his/her shares). The system has been depicted as a rectangular block containing a number of light-coloured rectangular blocks representing the structural elements of the system. The blocks have been related with each other through continuous lines showing their structural relationships. The example shows that system behaviour can be considered to be a collection of use-cases and a use-case can be seen as a unit of behaviour (behavioural element).

When a user initiates activation of a use-case, a number of related structural elements are activated. To indicate this, some of the rectangular blocks in Figures 3.4(b) and 3.4(c) are highlighted and connected through dashed arrows representing the information flow. The continuous arrow ending at a highlighted block
represents the input of a use-case, and the continuous arrow originating from a highlighted block represents the output of a use-case. Figure 3.4(b) shows the active blocks when the User Authentication use-case is being executed, and Figure 3.4(c) shows other blocks that become active when the Payment use-case is being executed.

Use-cases are usually expressed in a semi-formal way (referred to as use-case template or use-case notation) and include the following basic characteristics:

- A use-case name that uniquely identifies the use-case and clearly expresses its goal.
- A list of use-case actors, which are systems or persons that initiate interactions with the system to achieve their goals.
- A use-case trigger, which is the event (external, internal, or temporal) that causes initiation of the use-case.
- A list of use-case pre-conditions, which are the conditions (i.e., certain system states) that must be true for the trigger to meaningfully cause the initiation of the use-case.
- A list of use-case post-conditions, which are the desired system states after use-case completes.
- A list of use-case steps, which are either interactions between the system (or part of the system) and an actor, or references to other use-cases to which to delegate certain sub-goals [73].

For the purpose of system management, a use-case can be viewed as a description of a process in which a system receives a request, executes the use-case steps internally by means of the structural elements (e.g., system’s sub-systems, components), and produces a response. Based on this definition, and the fact that current use-case notations do not include information concerning the structural elements involved, the basic characteristics of use-cases are extended to include:

- A list of use-case structural elements, which are the system’s structural elements responsible for executing a number of use-case steps. Each use-case can be executed by a number of structural elements, and each use-case step is executed by exactly one structural element.

As the number of use-cases of complex distributed systems can be significant and the complexity of different use-cases can vary considerably, this thesis structures the use-cases. The following two sections discuss two approaches to structuring use-cases: use-case levels and use-case references.

### 3.5 Use-Case Levels

One of the important responsibilities of self-management of distributed systems is determining the root-cause of a system malfunctioning. Pinpointing a root-cause
in a system with a considerable number of complex use-cases running within a complex environment is one of the main challenges this thesis addresses. This thesis advocates that, for management purposes, use-cases can be best structured based on the software fault handling process performed in practice.

Enterprises have organised their software fault handling process around three teams, namely help desk support teams, system administrator teams, and system maintenance teams. As shown in Figure 3.5 (the fault handling process), when users encounter a problem regarding a specific system they first contact the related help desk support team and explain their problem. Based on the explanation of a problem, a help desk support team tries to determine the context (the top-level use-case) in which the problem occurred. This context information is passed to the system administrator team responsible for a set of servers on which different number of business processes run.

![Figure 3.5: Activities performed by a help desk support team, a system administrator team, and a system maintenance team consisting of functional analysts and system developers.](image)

The system administrator team tries to determine the business process responsible for the reported problem. If further investigation is needed, the information regarding the process is passed to the system maintenance team responsible for corrective and adaptive maintenance of the system. This team includes both functional analysts and system developers. The functional analysts try to determine the functional component responsible for the fault, and the system developers try to determine the root-cause of the problem in the code. After the root-cause of the problem has been determined, appropriate remedy actions are planned and performed. In line with the software fault handling process, this thesis proposes use-case levels to structure the use-cases of a distributed system.

To introduce use-case levels, further exploration of structural elements is needed. Structural elements are extensively explained in Section 5.2, for now it is sufficient
to know that, on the highest level, each distributed system is composed of a number of communicating sub-systems (in this thesis referred to as runnables), on the next lower level, each sub-system is composed of a number of components, and finally each component is composed of a number of classes. These elements (i.e., systems, runnables, components, and classes) are different types of structural elements. The following use-case levels for a distributed system are proposed [83, 85]:

1. **System level use-case** - From the end-user’s viewpoint, a system is, in fact, a black box and end-users are not interested in how the system executes the use-case. A system level use-case describes the external behaviour of a system. Assuming that the whole system is one big structural element, all use-case steps of a system level use-case are performed by one structural element (i.e., system). The use-cases in Figure 3.4 are all examples of system level use-cases. Help desk support teams are interested in this type of use-cases to narrow down the problem space at a very high level.

2. **Runnable level use-case** - From the viewpoint of a system administrator, only runnables and their connections are of importance. A runnable level use-case describes the internal behaviour of a system as interactions between runnables. All use-case steps of a use-case at this level are performed by runnables.

3. **Component level use-case** - From the viewpoint of a functional analyst, only components and their communications are of importance. A component level use-case describes the internal behaviour of a system as communications between components. All use-case steps of a use-case at this level are performed by components.

4. **Class level use-case** - From the viewpoint of a system developer, only classes, methods, and their invocations are of importance. A class level use-case describes the internal behaviour of a system as invocations between methods. All use-case steps of a use-case at this level are performed by classes/methods.

   The goal of the manual software fault handling process, described above, is to find the root-cause of system malfunctioning. Note that a root-cause is defined by each involved team in different terms (i.e., root-cause is a relative term). This thesis defines the root-cause of a system malfunctioning in terms of use-case steps and structural elements. The root-cause of malfunctioning during execution of use-case \( u_i \) is expressed as a pair \((s_k, us_j)\), where \( us_j \) is a use-case step belonging to \( u_i \), and \( s_k \) is the structural element executing the use-case step \( us_j \). As the types of use-cases and structural elements at each use-case level are different, each team determines its own root-cause of the reported problem. For example, the root-cause of the problem from the viewpoint of the system administrator team is a pair containing a runnable and a runnable level use-case step. From the viewpoint of the functional analyst, the root-cause is a pair containing a functional component and a component level use-case step. With respect to structural elements, the granularity
of the root-cause of the reported problem can vary from coarse-grained (faulty system) to less coarse-grained (faulty process), to fine-grained (faulty component), and finally to more fine-grained (faulty code).

Introducing the above use-case levels has the following advantages concerning management of distributed systems:

- Use-case levels facilitate knowledge acquisition. Acquiring domain knowledge necessary for model-based\textsuperscript{4} approaches is known to be difficult [42, 150]. Self-management of distributed systems requires knowledge about a managed system, and different domain experts (such as help desk support team, system administrators, etc.) are the providers of this knowledge. As use-case levels reflect the view of different domain experts, the required knowledge about use-cases of a specific level can be independently and conveniently specified in use-case notation by the corresponding domain experts.

- Use-case levels provide an opportunity for autonomic managers to pinpoint the root-cause of a system malfunctioning at different granularity levels (i.e., system, runnable, component, and class).

- Use-case levels divide a problem space into a number of sub-spaces each with their own type of problems. For example, problems such as a broken connection, an incorrect start-up sequence of processes, or excessive heap usage can occur at runnable level, problems such as incorrect component version, or unregistered component can occur at component level, and problems such as incorrect parameters, or uninitialised class members can occur at class level. This means that an autonomic manager controlling the correct execution of a use-case at a specific level has to pay attention only to the specific category of problems occurring at that particular level.

3.6 Use-Case References

To facilitate the diagnostic task of autonomic managers, the previous section introduced use-case levels to structure the knowledge needed to diagnose system malfunctioning. Note, however, that each of these use-cases, independent of the level at which it resides, may reference other use-cases. Two types of use-case references are distinguished: horizontal and vertical references. The following sections explain both types of use-case references in relation to the use-case levels.

3.6.1 Horizontal References

Recall that a use-case contains a number of use-case steps. Each use-case step can be either an activity or a reference to another use-case to delegate a certain sub-goal to be realised. A reference from a use-case step (of a use-case at a specific level) to a use-case at the same level is called a \textit{horizontal reference}.

\textsuperscript{4}The self-management approach in this thesis uses a model of an existing distributed system to perform management tasks.
Figure 3.6 shows the different use-case levels and horizontal references between use-cases at the same level. The ovals at different use-case levels represent behaviour elements\(^5\) (BE\(_i\)) each containing one or more blocks representing the relevant structural elements (SE\(_i\)). All structural elements within one use-case level are of the same type (according to the definition of a use-case level presented in Section 3.5). For example, at the runnable level, structural elements SE\(_2\), SE\(_3\), and SE\(_4\) are of type runnable. Use-case BE\(_3\) is executed by structural elements SE\(_2\), SE\(_3\), and SE\(_4\), and use-case BE\(_4\) is executed by structural elements SE\(_3\) and SE\(_4\). Note that different behaviour elements within the same use-case level may be executed by the same structural elements (both BE\(_3\) and BE\(_4\) use SE\(_3\) and SE\(_4\) to be executed).

Use-case steps, including references to other use-cases, are executed by structural elements. A dashed line in the figure, originating from a structural element (SE\(_i\)) and ending at a behaviour element (BE\(_i\)), represents the communication between use-cases at the same level. For example, at the runnable level, the dashed line from SE\(_3\) to BE\(_4\) shows that the use-case step (horizontal reference) running on SE\(_3\) is an invocation of BE\(_4\) at the same level. An example of a horizontal reference is given in the next section (see Figure 3.12 - step (2)).

3.6.2 Vertical References

A reference from a use-case step of a use-case at one level to another use-case at a different level is called a vertical reference. There are two types of vertical references: a use-case reference from a higher level use-case to a lower level use-case,

\(^5\)The terms use-case and behavioural element are used interchangeably.
called a **downward vertical reference**, and a use-case reference from a lower level use-case to a higher level use-case called an **upward vertical reference**. Figure 3.7 shows the use-case levels with the same behavioural and structural elements as Figure 3.6. The difference is that Figure 3.7 depicts the vertical references including both downward and upward.

A dashed line from a higher level toward a lower level in the figure represents downward communication between use-cases at the different levels. There is a compositional relationship between a structural element executing the downward reference of a use-case at a higher level, and a number of structural elements executing the use-case steps of the referenced use-case at the lower level. For instance, structural element SE2, at the runnable level, consists of structural elements SE5, SE6, and SE7 at the component level. When the downward reference executes on SE2, the system switches to the execution of use-case BE5 executed by the components SE5, SE6, and SE7.

The figure also illustrates that different structural elements used by the same behaviour can have different compositions. For instance, component level use-case BE5 uses components SE5, SE6, and SE7. As there are two downward vertical references executed by SE6, and SE7, there are two different compositional relationships: component SE6 only consists of the classes SE8 and SE9 (and not SE10), however, component SE7 consists of the classes SE8, SE9, and SE10. Note that the two classes SE8 and SE9 are used by both structural elements (components) SE6 and SE7.

The downward referential relationship between behavioural elements makes it possible for autonomic managers to narrow down the search space of the root-cause of a system malfunctioning from complex behaviours to simpler behaviours, by following the line of references from a higher level toward lower levels. The
compositional relationship between structural elements makes it possible for autonomic managers to pinpoint more fine-grained structural elements as the faulty elements, by following the same line of references. An example of a downward vertical reference is given in the next section (see Figure 3.12 - step (6)).

A dashed line from a lower level to a higher level represents upward communication between use-cases at the different levels. For instance, use-case SE8, at the class level, needs a functionality of a library (component) that is executed by SE6 at the component level. The invocation statement executed within SE10 is an upward reference. Note that there is no compositional relationship between a structural element executing upward reference of a use-case at a lower level, and the structural elements executing the use-case steps of the referenced use-case at the higher level. An example of an upward vertical reference is given in the next section (see Figure 3.18 - step (2)).

Figure 3.7 illustrates only behavioural relationships between two neighbouring levels. Nevertheless, to cope with some practical situations, the definition of the vertical reference also allows behavioural relationships between non-neighbouring levels. The invocation of a method of a class from the main method of a runnable (a downward vertical reference from the runnable level use-case to the class level use-case), and the invocation of an executable from a method of a class (an upward vertical reference from the class level use-case to the runnable level use-case) are examples of such practical situations.

Remember that the objective of this thesis is to achieve self-management of distributed systems by managing the different behaviours of these systems. To manage an individual behaviour, an autonomic manager is made responsible for managing a use-case representing a behavioural element. To manage the behaviour of a whole system, the autonomic managers of all of the behavioural elements are related to each other according to the referential relationships (both horizontal and vertical). These autonomic managers make the self-management of a distributed system possible. An example scenario in the following section illustrates use-cases, use-case levels, and their relationships.

### 3.7 An Example Scenario

This section presents a user authentication scenario for a simplified version of a complex portal system, with possible causes of failure at different use-case levels. The following scenario illustrates how enterprises authenticate business users requesting access to a company’s secure portal system. Portal systems typically integrate a number of legacy systems, presenting them on the web as a single system. Consequently, the authentication logic for the system as a whole is usually spread out and embedded in different sub-systems of a portal system including legacy sub-systems. Figure 3.8 illustrates the activities of the example scenario. These activities are realised by cooperation of different structural elements (at runnable level) shown in Figure 3.9.
To access a portal system, business users provide their certificates to the AccessManager sub-system using a negotiation process, established between the user’s Browser and the AccessManager (see Figure 3.9). Upon receiving the user’s certificate, the AccessManager verifies the certificate, and passes it to the BusinessIntegrator sub-system using an appropriate connection. The BusinessIntegrator communicates with the DatabaseManager sub-system, extracts the user’s identity (userid), acquires the password for the given userid, constructs login information (userid/password), and sends it to the BusinessManager sub-system (legacy back-end). The BusinessManager authenticates the user’s credential and returns the result of the authentication to the BusinessIntegrator. Finally, the BusinessIntegrator passes the result of the authentication through the AccessManager back to the user’s Browser.

The above description of the authentication process assumes nothing goes wrong (correct behaviour). Suppose, however, that an error occurs: a user is denied access even though he/she has a valid certificate and identity. The root-cause of this malfunctioning needs to be discovered. The root-cause might be somewhere in the system code. It is obvious that pinpointing such a root-cause in a system with thousands of lines of code distributed on different physical sub-
3.7 An Example Scenario

systems is not straightforward. Distinguishing a hierarchy of levels in the structure and the behaviour of a system on the basis of use-case descriptions, and relating root-causes to these levels, can facilitate the diagnostic task.

Based on the definition of a root-cause (see Section 3.5), the root-cause of system malfunctioning at a specific use-case level relates to the structural element dedicated to that level. Therefore, the root-cause of malfunctioning of a system level use-case relates to the whole system, a runnable level use-case relates to a runnable, a component level use-case relates to a component, and a class level use-case relates to a class/method. The above mentioned levels of use-case descriptions are clarified in the following sections together with illustrations of level-related errors.

3.7.1 System Level View

Usually an end-user informs the help desk support team about a system malfunctioning by reporting ‘system x does not work’. To narrow down the search space of possible root-causes, the first question the help desk support team asks is: ‘What were you doing?’ or more specifically: ‘Which behaviour of system x were you using?’ As mentioned before, a system can contain many behavioural elements (such as User Authentication, Trade Entry, Payment, and Update Shares introduced in Section 3.4) that divide the behaviour space of a system into the top level (system level) use-cases. By associating an autonomic manager with each system level behavioural element, the search space of possible root-causes of a system malfunctioning is divided between multiple autonomic managers. Therefore, the user-system interactions regarding one specific behavioural element can provide the starting point for discovering the root-cause of the mentioned system malfunctioning.

Figure 3.10 shows the authentication process (behavioural element) from the viewpoint of a user, described as a system level use-case. Note that the use-case steps have been intentionally formulated in terms of user and system. A system malfunctioning might be caused by the fact that a user did not provide a certificate (step (2)), the system was not able to authenticate the user (step (4)), or the system did not show the authentication result (step (5)). The help desk support teams are interested in this use-case level to determine the use-case step that caused the undesired situation.

Figure 3.11 shows the structural element a user sees during interaction with the system. The system is the only structural element involved at this level. Note that the root-cause of a malfunctioning at system level is coarse-grained. More fine-grained root-causes are deeper in the hierarchy of use-case levels.

3.7.2 Runnable Level View

Step (4) in Figure 3.10 is the invocation of the Authentication Realisation use-case at runnable level. Figure 3.12 refines the mentioned use-case step, and shows the Authentication Realisation use-case as interactions between runnables, from the
name: User Authentication
actors: user
trigger: requesting access to portal system
pre-conditions: user has a valid certificate
post-conditions: user is authenticated
structural elements: System
steps:
1. System receives an access request from user,
2. System requests user to provide certificate,
3. System receives user’s certificate,
4. System calls Authentication Realisation to authenticate user,
5. System shows authentication result to user.

Figure 3.10: User Authentication use-case at system level.

viewpoint of a system administrator. Note that each runnable, as mentioned in Figure 3.9 (Browser, AccessManager, etc.), may implement one or more use-case steps.

name: Authentication Realisation
actors: Browser sub-system
trigger: requesting access to portal system
pre-conditions: user has a valid certificate
post-conditions: user is authenticated
structural elements: Browser, AccessManager, BusinessIntegrator, DatabaseManager, BusinessManager
steps:
1. Browser passes user’s certificate to AccessManager,
2. AccessManager calls Certificate Verification to verify certificate,
3. AccessManager passes certificate to BusinessIntegrator,
4. BusinessIntegrator requests a database session from DatabaseManager,
5. DatabaseManager retrieves database session,
6. BusinessIntegrator calls Auth-info Preparation to prepare login info,
7. BusinessIntegrator delegates login info to BusinessManager,
8. BusinessManager authenticates the user,
9. BusinessManager passes result to BusinessIntegrator,
10. BusinessIntegrator passes result to AccessManager,
11. AccessManager passes authentication result to Browser.

Figure 3.12: Authentication Realisation use-case at runnable level.

Figure 3.13 shows which use-case steps are executed by which runnable, and the remote connections between runnables. At this level, use-case steps are more specific, and the coarse-grained structural element (system) has been decomposed into a number of finer-grained structural elements (runnables). An autonomic manager at this level can pinpoint a specific runnable or the connection between two runnables as the root-cause of the mentioned malfunctioning.
As mentioned before, one of the most frequently occurring errors at the runnable level is a broken connection. The autonomic manager responsible for the Authentication Realisation monitors use-case steps (such as step (3) in Figure 3.12) that represent the communication between two runnables to detect such errors. The autonomic manager checks whether the connection (in this case between the AccessManager and the BusinessIntegrator) is functioning correctly.

The runnable level Certificate Verification use-case illustrates horizontal use-case relationships (see Figures 3.14 and 3.15). This use-case refines step (2), the horizontal reference, of the Authentication Realisation use-case (see Figure 3.12), and describes how the AccessManager delegates verification of the user’s certificate to the CertificateVerifier. Upon receiving the certificate, the CertificateVerifier inspects the validity of the certificate by comparing it to the certificates in its database. If a match is found then the CertificateVerifier sends the certificate to the CertificateAuthority to validate its expiration date. Finally, the CertificateVerifier sends the verification result back to the AccessManager.

Presenting the Certificate Verification use-case clarifies that different use-cases, at the same use-case level, can be executed by different structural elements. The runnables Browser, BusinessIntegrator, BusinessManager, and DatabaseManager are not shown in Figure 3.15 as none of the use-case steps of the Certificate Verification are executed by the runnables. The autonomic manager responsible for the Certificate Verification use-case does not need to care about the management of those runnables.

Step (6) in Figure 3.12 is an example of a downward vertical reference, from
3.7.3 Component Level View

Figure 3.16 refines step (6) of the Authentication Realisation use-case, and shows the Auth-info Preparation use-case from the viewpoint of a functional analyst in which only components and their interactions are of importance. Each component performs one or more use-case steps.

- **name:** Auth-info Preparation
- **actors:** BusinessIntegrator sub-system
- **trigger:** passing certificate
- **pre-conditions:** existence of a certificate
- **post-conditions:** login-info (userid/password) is prepared
- **structural elements:** CertificateParserComp, PrepareAuthComp
- **steps:**
  1. CertificateParserComp receives certificate,
  2. CertificateParserComp extracts userid,
  3. CertificateParserComp passes userid to PrepareAuthComp,
  4. PrepareAuthComp receives userid,
  5. PrepareAuthComp calls Login-info Retrieval to prepare login info,
  6. PrepareAuthComp returns login info.

Figure 3.16: Auth-info Preparation use-case at component level.

Runnable BusinessIntegrator contains two components: the CertificateParserComp, that parses a certificate and extracts a user identity, and the PrepareAuthComp, that constructs login information. Figure 3.17 shows these components and the use-case steps executed by each.
3.7 An Example Scenario

Use-case step (3) in Figure 3.16 expresses the communication between the two components \textit{CertificateParserComp} and \textit{PrepareAuthComp}. The autonomic manager responsible for the \textit{Auth-info Preparation} use-case monitors the mentioned use-case step by keeping track of the state of the current version of a component. Monitoring is needed because the components might be incompatible, the version numbers of the components might not correspond.

3.7.4 Class Level View

Figure 3.18 refines step (5) of the \textit{Auth-info Preparation} use-case, and shows the \textit{Login-info Retrieval} use-case from the viewpoint of a system developer in which only classes, methods, and their interactions are of importance. Note that each class or method may perform one or more use-case steps.

<table>
<thead>
<tr>
<th>name: Login-info Retrieval</th>
</tr>
</thead>
<tbody>
<tr>
<td>actors: PrepareAuthComp component</td>
</tr>
<tr>
<td>trigger: passing userid</td>
</tr>
<tr>
<td>pre-conditions: existence of a userid</td>
</tr>
<tr>
<td>post-conditions: login-info (userid/password) is retrieved</td>
</tr>
<tr>
<td>structural elements: AuthInfoClass, RetrieveAuthInfo</td>
</tr>
<tr>
<td>steps:</td>
</tr>
<tr>
<td>(1) AuthInfoClass receives userid,</td>
</tr>
<tr>
<td>(2) RetrieveAuthInfo calls Data Retrieval to retrieve password from database,</td>
</tr>
<tr>
<td>(3) RetrieveAuthInfo constructs login info,</td>
</tr>
<tr>
<td>(4) RetrieveAuthInfo returns login info.</td>
</tr>
</tbody>
</table>

Figure 3.18: \textit{Login-info Retrieval} use-case at class level.

The \textit{PrepareAuthComp} component contains a class called \textit{AuthInfoClass}. The constructor method of this class receives the userid and saves it for future use. This class also contains a method called \textit{RetrieveAuthInfo} which uses the established connection with the database to retrieve user related information from the database.

Figure 3.19 shows this class and its methods, and the use-case steps executed by each of the methods. One of the errors that can occur is that either the \textit{DatabaseManager} is down or the \textit{DatabaseManager} provides incorrect information about a user’s identity. The autonomic manager responsible for the \textit{Login-info Retrieval} monitors the upward vertical reference expressed in use-case step (2) in Figure 3.18 to determine these errors.

This step refers to the \textit{Data Retrieval} use-case defined at the component level (see Figure 3.20). The use-case describes the communication between the \textit{DataRetrievalComp} and the \textit{JdbcComp} components. The \textit{DataRetrievalComp} hides the persistence mechanism used by the \textit{BusinessIntegrator}, and requests the \textit{JdbcComp} to retrieve the user-record (userid/password) from the specified database table.

Note that if the result returned from \textit{DatabaseManager} is incorrect then a \textit{NullPointerException} would occur during the construction of the login information (step (3) in Figure 3.18), and the execution of the use-case would have terminated abnormally. Section 5.1 explains how these events are modelled.
3.8 Summary

This chapter argues that self-management of complex distributed systems must deal with both the behavioural and structural complexities of these systems, and that the behavioural element is the preferred unit of management for autonomic managers. As an autonomic manager requires knowledge on its managed unit, the chapter advocates to reuse the knowledge on a system’s behaviour and structure, acquired during the software design and development phase. This knowledge is mainly specified in use-cases and related UML diagrams. Use-case notations were extended to also include structural information.

This chapter explains how use-cases can be structured and organised in use-case levels, including references, based on an existing software fault handling process. Finally, the chapter describes an example scenario illustrating use-cases at different levels: system, runnable, component, and class level. The scenario shows how a system’s behaviour at a specific level can be related to the system’s software architecture at that level. Moreover, the scenario shows how the granularity of the root-cause of problems from higher levels to lower levels changes from coarse-grained to fine-grained.
Chapter 4

The Management Model

To face the complexity of management of distributed systems, the previous chapter advocated the use of multiple autonomic managers, one for each behavioural element. The relationships between behavioural elements determine the relationships between autonomic managers. This chapter explains how an autonomic manager manages a behavioural element (i.e., use-case) of a system, and how autonomic managers cooperate with each other to manage the behaviour of the whole system.

The self-management approach described in this thesis is a model-based approach: the important properties of distributed systems are abstracted, and the main entities that play a role in the management of these systems are identified. This chapter presents the management model: the model of individual autonomic managers and their interactions.

Section 4.2 zooms in on autonomic managers and behavioural elements, and provides an overview of their internal structures. Sections 4.3 through 4.6 explain the autonomic manager’s main entities (analyser, diagnoser, planner, and plan translator), their working, and their role in the management of system malfunctioning. These entities are formally specified in Chapter 6. Section 4.7 describes how an autonomic manager coordinates the working of its main entities, and cooperates with other autonomic managers to realise the management of the whole system. Section 4.8 depicts the internal structure of information flow entities (sensor, symptom, hypothesis, plan, effector) that are passed from one main entity of the autonomic manager to the next.

The autonomic manager and its main entities utilise (logical) rules to determine the occurrence of symptoms, identify diagnoses, select proper plans, and translate plans to executable adaptation instructions. There are two sets of rules: generic rules and domain-specific rules. The domain-specific rules are relevant to the management of a specific distributed system. In contrast, the generic rules are domain independent. They are relevant to the management of all distributed systems supported by the framework proposed in this thesis. The generic rules are the pre-defined part of a management model and domain-specific rules are added as the result of interactions with domain experts. Appendix A illustrates and explains the generic part of these rules.
4.1 Introduction

Computer system malfunctions occur in every enterprise. Because of the inability of human beings to produce error-free software, software malfunctioning is almost inevitable. According to a technical report published by NASA [180], ‘[...] it is extremely difficult to produce a flawless piece of software. For humans, perfect knowledge of a problem and its solution is rarely achieved. Even if a programmer has sufficient knowledge to solve a problem, that knowledge must be transformed into a systematic representation adequate for automatic processing’. Hard-to-detect design faults are likely to be introduced during development, especially, when a software program is developed by multiple development teams (and vendors), and maintained by different administrator teams. System malfunctioning can occur if different teams misunderstand each other about even one small issue.

Developers and system administrators all encounter system malfunctions in the course of time. Once a root-cause is identified software is most often repaired manually. The knowledge gathered during this process is often stored in a human language in a knowledge repository. Human languages are inherently ambiguous and can be interpreted in different ways. Expressing this knowledge in a formal language and embedding the human diagnostic process into a separate software program makes it possible for both human (developers, system administrators, etc.) and automated systems to reuse this knowledge for problem determination and repair actions. This thesis proposes a management model in which this knowledge plays an important role.

4.2 Management Model Overview

Recall from Chapter 3 that autonomic management of a distributed system is performed by multiple autonomic managers. System behaviour specified in use-cases is the unit of management, and each use-case’s execution is managed by a dedicated autonomic manager. Management of execution of a use-case is a complex task. To cope with this complexity, and to achieve a reasonable degree of maintainability, transparency, extensibility, and adaptability, an explicit design decision has been made to define separate entities within an autonomic manager for analysis, diagnostic, planning, and plan translation processes. Also, the relationships and interfaces between these entities are formally defined to allow them to be flexible in changing their internal algorithmic and computational approach.

The following sections provide a high level overview of the autonomic manager, managed use-case, and their relationships and thereafter describe a scenario regarding an abnormal behaviour of a distributed system.

4.2.1 High Level Overview

This section zooms in on the ‘autonomic manager’ and ‘behavioural element’ depicted in Figure 3.3(b), providing an overview of their internal structures. Figure 4.1 depicts the management feedback process, inspired by the autonomic com-
puting architectural blueprint, in more detail. At the highest level, the figure shows an autonomic manager and a behavioural element (i.e., managed use-case). The managed use-case is depicted as an oval containing a number of rectangular blocks representing structural elements on which use-case steps are executed. These structural elements are extended with sensors and effectors. Sensors provide runtime information from a managed use-case to an autonomic manager, and effectors implement adaptation instructions from an autonomic manager to a managed use-case.

Each autonomic manager has four main entities: Analyser, Diagnoser, Planner and Plan Translator. These entities share the knowledge available in the structural and behavioural models regarding the managed use-case. The analyser is responsible for identifying abnormal behaviour (symptom) of a managed use-case based on values received by the sensors and the information available in the shared knowledge-base. The diagnoser determines the root-cause (diagnosis) of the abnormal behaviour identified. Based on this diagnosis, the planner constructs remedy plans and passes the plans to the plan translator to be translated into instructions for the effectors instrumented in the code of the managed use-case. Note that the autonomic manager’s main entities in the architecture depicted in Figure 4.1 are slightly different from the ones in the autonomic computing architectural blueprint depicted in Figure 1.3.

The monitor entity of the architectural blueprint in Figure 1.3 is replaced in Figure 4.1 by Analyser. Analyser not only monitors information but also infers new information. The analyser entity of the architectural blueprint in Figure 1.3 is replaced in Figure 4.1 by Diagnoser. Diagnoser does not analyse the information regarding the managed use-case but it uses meta-knowledge to diagnose the cause of abnormal behaviour. The execute entity of the architectural blueprint in Figure 1.3 is replaced in Figure 4.1 by Plan Translator. Plan Translator does not
directly execute plans but it translates them into instructions to be executed by effectors.

Recall from Chapter 3 that an existing distributed system usually has a considerable number of complex use-cases running within a complex environment. In this thesis, the complexity of the self-management of distributed systems is reduced by dividing the use-cases space into use-case levels and relating use-cases to each other at various levels according to their references. The proposed management model provides a mechanism to support communication and cooperation between autonomic managers.

### 4.2.2 A Failure Scenario

The working of the main entities (such as analyser) of an autonomic manager in the management model is further explained in the next sections by a failure scenario. This scenario concerns a failure of the Trading System, introduced in Chapter 3, during execution of the Payment use-case. This section describes the scenario.

Naming plays an important role in distributed systems (see [177] for an extended discussion about naming). Comparable to a card catalog, that maps names of books to their location in a library, a naming service associates names with locations of services, and maintains a set of bindings (associations between names and locations). Usually one sub-system (a server) registers a service with a naming service, and another sub-system (a client) uses a naming service to locate that service by name, to retrieve it, and to use it for its own purposes.

![Interaction Diagram](image)

**Figure 4.2:** The interaction diagram between different structural elements used in the naming service example. A rectangular block represents a structural element of a system. An interaction is depicted by an arrow, and the text above an arrow depicts a method invocation. The star on the arrow at the bottom indicates the occurrence of a malfunctioning.

Assume the Payment use-case of the Trading System is managed by an autonomic manager. The Payment use-case involves interaction between a client
4.3 Analyser

The main responsibility of an Analyser is to determine whether one or more symptoms have been detected, based on the information returned by the relevant sensors. The entities related with the analyser are described below. The two information flow entities, Sensor and Symptom, are explained in detail in Sections 4.8.1 and 4.8.2 respectively.

Entities Related to Analyser

An Analyser utilises one or more SymptomOccurrenceRules which are logical combinations of comparison operators over sensor values. One SymptomOccurrenceRule uses one or more sensors instrumented in the code of a managed use-case (Job\(^2\)). There are at least three sensors associated with a job: two inform the autonomic manager about the beginning and end of job execution, and the third monitors a use-case step\(^3\). Figure 4.3 shows the entities and relationships involved in ER diagram form.

An Analyser’s activities are affected by a set of policy rules represented as AnalyserStrategicRules. For example, consider a distributed system consisting of

---

\(^1\)Entity-Relationship (ER) diagrams, originally proposed in 1976 by Chen [36], are used in this thesis to describe all entities, relationships, and attributes needed to implement the management feedback loop described in the previous section. UML state diagrams [69] are used to describe the activities of the entities and the interactions between them including their information flow. For the sake of readability, model elements are denoted in the Courier font, names of model entities start with an upper case letter, and names of relationships and attributes start with a lower case letter.

An ER diagram depicts the static structure of an entity. In an ER diagram, entities are represented by rectangles, and relationships are represented by diamonds. Each entity contains a number of attributes describing the characteristics of the entity. Attributes are represented by ellipses. A range of numbers (or just one number) along continuous lines indicate cardinality which expresses the number of relationships allowed per entity.

\(^2\)The Job entity is an element of the system behavioural model, and is described in Section 5.1.

In this thesis, the terms use-case and job are used interchangeably.

\(^3\)The assumption is that each use-case contains at least one use-case step.
Figure 4.3: The ER diagram depicts the relationships of an analyser with its symptom occurrence rule and policy rule. Furthermore, the diagram depicts that the symptom occurrence rule uses sensors, instrumented in the code of a managed use-case (job), to determine a symptom occurrence.

of two sub-systems that communicate with each other using message delivery middleware. Also assume that the middleware is always unavailable between 00:00 AM and 01:00 AM. If the two sub-systems attempt to communicate during this period, the MessageDeliveryFailed symptom occurs. The occurrence of this symptom can be ignored by the analyser if the following policy rule is included in the AnalyserStrategicRules: ‘if the MessageDeliveryFailed symptom is inferred between 00:00 AM and 01:00 AM then the symptom occurrence should be ignored’.

Analyser’s Process

The process model of Analyser is shown in Figure 4.4 in UML state diagram form. When Analyser is activated (by the autonomic manager), it executes its policy (strategic) rules, and subsequently tries to infer one or more symptoms, based on the observed values coming from sensors, using its SymptomOccurrenceRules.

Analyser’s Role in the Scenario

To analyse the situation around the abnormal behaviour mentioned in the failure scenario described above, the analyser deploys the following entities specified by domain experts: PaymentServiceSensor, ServiceRetrievalFailed, and a set of SymptomOccurrenceRules. When an autonomic manager obtains a value from PaymentServiceSensor, it sends this value to the analyser. The analyser uses the knowledge specified as SymptomOccurrenceRules to identify abnormal behaviour. An example of a SymptomOccurrenceRule is: ‘if the value returned by PaymentServiceSensor is null then the ServiceRetrievalFailed symptom has occurred’.

---

4A state diagram depicts the dynamic behaviour of an entity, and consists of a collection of states and transitions between states. There are two special states: initial state and final state. An initial state is the state that an entity is in when it is first activated, and a final state is the state from which no transition is allowed. A transition is a progression from one state to another, and is triggered after an activity takes place. To keep the diagram simple, a number of related states together with their transitions can be grouped as one composite state.
4.4 Diagnoser

The main responsibility of a Diagnoser is to reason about root-causes of the detected symptoms and to determine one or more diagnoses. A diagnosis is determined on the basis of a pre-defined set of hypotheses by utilising a number of sets of rules belonging to the diagnoser. A possible root-cause of detected symptoms is referred to as a hypothesis.

Note that the occurrence of different symptoms can often be explained by the same hypothesis, and different hypotheses can explain the same symptom. Determining proper diagnosis is not straightforward. A diagnoser needs to deal with incomplete and inconsistent knowledge. For instance, when the diagnoser is informed about the ‘failure to read a file from a shared drive’ but it is not informed about the ‘broken network connection’. Moreover, there can be a generality relation between hypotheses. For example, the `FileSystemNotAvailable` hypothesis is more generic than the `DirectoryNotReadable` hypothesis. The proposed diagnostic model deals with these issues.

The diagnostic model has been inspired from the model, introduced by Brazier et al. [24], where the authors describe a diagnostic reasoning process. The process starts by selecting hypotheses on which to focus, validating these hypotheses by determining relevant observations to be performed, and evaluating the hypotheses on the basis of the observation results. This approach combines causal and anti-causal domain knowledge for diagnostic reasoning processes. In the first case, derivations follow the direction of causality, that is, symptoms are derived from hypotheses (possible causes). In the second case, the direction of derivation is against the direction of causality, that is, the reasoning process proceeds from actual occurred symptoms to derive hypotheses [26].
Brazier et al. applied their diagnostic model to the diagnosis of chemical processes and soil sanitation. This thesis applies their diagnostic model to the domain of software failure diagnosis. The reasons for adopting their model are:

1. their model is designed to reason about incomplete and inconsistent knowledge: whether or not a symptom has occurred may be specified as unknown. The value of a symptom can be unknown if the relevant sensors of the symptom have been instrumented in the code at points that are not executed. The value of a symptom can also be unknown if the relevant sensors of the symptom have not been instrumented in the executable code of a managed use-case (e.g., a symptom that represents a mis-configured property in a configuration file, a symptom that represents a broken network connection, etc.). Note that, during execution of a use-case, only sensors are triggered that have been instrumented in the code at points that are executed. Consequently, the occurrence of only a part of symptoms becomes known.

2. the nature of knowledge specification needed for diagnostic reasoning is declarative. Declarative knowledge can be provided by non-technical domain experts, can be stored and maintained in a file separate from the code for the diagnoser, and it can be modified (to some extent) during runtime without redeploying the autonomic manager.

In the following, the entities related to the diagnoser are described, and it is explained how these entities work together to realise the diagnostic process. Section 4.8.3 explains the information flow entity, Hypothesis, in detail.

Entities Related to Diagnoser

Figure 4.5 depicts the relationships of Diagnoser with different sets of rules. How and in which order these different sets of rules are executed by the diagnostic engine are explained later. This section explains the role of each rule set with respect to a diagnostic task.

Rules in HypothesisSelectionRules are required for selecting a hypothesis from a set of hypotheses. A selection rule selects a hypothesis for validation,
4.4 Diagnoser

based on one or more symptoms. Note that there is a possibility that none of the selection rules succeed to select a hypothesis. Meta-knowledge is then needed to determine the next action. Diagnoser’s activities are influenced by a set of policy rules represented as DiagnoserStrategicRules. An example of such a heuristic policy rule is: ‘if more than two relevant symptoms of a hypothesis are unknown then do not perform an inspection concerning the occurrence of these symptoms.’

Rules in HypothesisValidationRules validate a selected hypothesis by checking whether all of the symptoms of the selected hypothesis are known to have occurred. To enable a diagnoser to observe the real world for the occurrence of a symptom, an InspectivePlan is associated with each symptom. An InspectivePlan consists of a collection of actions executed by the diagnoser to explore whether or not a related symptom has occurred.

A Diagnoser also has a set of rules (ChildResultToSymptomRules) to translate results of child autonomic managers into Symptoms. The result of the autonomic process of an autonomic manager contains the diagnosis determined by its diagnoser. A child autonomic manager delivers its diagnosis to its parent. To enable a parent to incorporate diagnostic knowledge of its children, a diagnoser of a parent autonomic manager combines the diagnoses of all children and translates them into symptoms using the ChildResultToSymptomRules. For example, if the diagnoses of two children of the same autonomic manager are directoryNotWritable and directoryNotReadable then a ChildResultToSymptomRule can conclude the occurrence of filesystemNotAvailable symptom.

Rules in HypothesisEvaluationRules are required to assess (reject or accept) valid hypotheses based on certain criteria such as their pre-defined importance or generality. Finally, rules in DiagnosisDeterminationRules are required to determine a diagnosis based on the result of the assessment of the selected hypothesis.

Diagnoser’s Process

A diagnostic process starts with the anti-causal knowledge (symptoms) to derive which hypotheses are to be validated. Causal knowledge is then used to determine the occurrence (or absence) of certain symptoms in the past. If the occurrence (or absence) of those symptoms is not known then the diagnostic engine tries to determine their occurrence. This section describes the state diagram of the diagnostic process including interaction between hypothesis selecting, validating, evaluating, and determining states. How exactly a hypothesis is selected, validated, evaluated, and determined is explained in the Appendix sections A.2 through A.5 that describe the generic hypothesis rules.

Figure 4.6 shows the working of a Diagnoser. A diagnoser is activated by either its autonomic manager or by arrival of diagnostic information from one of the children of the autonomic manager. After activation, the diagnoser executes its policy (strategic) rules. If a diagnoser is activated by its autonomic manager, it immediately activates its HypothesisSelectionRules. Otherwise, it first maps the diagnostic information of its child into certain symptoms, and thereafter goes to the SelectingHypotheses state.
From the collection of hypotheses that indicate a possible root-cause of problems related to a managed use-case, **HypothesisSelectionRules** selects one or more hypotheses as possible root-causes of the occurred symptoms. The criteria for selecting a hypothesis from the collection are determined by meta-knowledge. If every attempt to select a hypothesis fails (for example, because all hypotheses have already been examined), a diagnoser transits to the **DeterminingDiagnosis** state, and performs one of the following activities:

1. If there are one or more assessed (accepted or rejected) hypotheses then the diagnoser activates the **DiagnosisDeterminationRules** to infer one or more diagnoses. After that, the diagnoser terminates, whether the execution of the **DiagnosisDeterminationRules** has led to any diagnosis or not.

2. If no hypotheses have been assessed, a diagnoser immediately terminates without determining a diagnosis.

The transition from the **SelectingHypotheses** state to the **ValidatingHypotheses** state occurs if at least one hypothesis can be selected by a **HypothesisSelectionRule**. The diagnoser activates the **HypothesisValidationRules** to determine...
whether the values of all symptoms of a selected hypothesis are known. A hypothesis is said to be valid if the occurrence of all of its associated symptoms are known or it is known that they have not occurred. A hypothesis is considered to be invalid if at least one of its symptoms is unknown. If a hypothesis is valid then the diagnoser activates the HypothesisEvaluationRules to accept or reject this hypothesis as being the possible root-cause of a system malfunctioning. Immediately after assessment, the diagnoser returns to the SelectingHypotheses state.

If the occurrence of at least one of the symptoms of a hypothesis is unknown and there is an InspectivePlan associated with that symptom then the diagnoser starts the execution of the InspectivePlan to inspect the occurrence of the related symptom. The execution of an inspective plan is managed by one of the children of the autonomic manager performing the diagnostic task. The information acquired by the execution of the inspective plan is delivered to the parent autonomic manager.

### Diagnoser’s Role in the Scenario

To determine a diagnosis for the abnormal behaviour mentioned in the failure scenario, the autonomic manager requests its diagnoser to investigate the root-cause of the failure. Note that the analyser has already inferred the occurrence of the ServiceRetrievalFailed symptom. The knowledge the diagnoser needs concerns the possible root-causes (hypotheses) and their relationships with the symptoms.

Two hypotheses are defined by the domain expert as possible root-causes of the ServiceRetrievalFailed symptom: NamingServiceDown and NameNotRegistered. The first one declares that NamingService is down, and the second one states that no binding can be found between the given service name and a location.

The diagnoser should be first sure about the value of all symptoms. Suppose the domain expert defines the following rules:

- ‘if both ServiceRetrievalFailed and NSNotAccessible symptoms have occurred then a possible root-cause is NamingServiceDown’.
- ‘if both ServiceRetrievalFailed and NamingConflict symptoms have occurred then a possible root-cause is NameNotRegistered’.

When the diagnoser tries to execute the first rule, it detects that it does not know whether the NSNotAccessible symptom (describing that NamingService is not accessible for PaymentClient) has occurred or not. The diagnoser needs to have knowledge of how to check the accessibility to proceed. This knowledge is represented by an inspective plan that consists of a collection of actions to be executed by the diagnoser. The inspective plan associated with the NSNotAccessible symptom is defined by the domain expert as a plan (called CheckNSAccessibility) containing only one action to send a ping message to NamingService to see whether it responds. Figure 4.7 illustrates the relation between the mentioned hypotheses, symptoms, and inspective plans.

The diagnoser also detects that the occurrence of the NamingConflict symptom (describing that PaymentClient and PaymentServer have different names for the
same service) is unknown. The inspective plan associated with the NamingConflict symptom is defined as a plan (called CheckNamingConflict) containing three actions: (1) retrieve the service name from the configuration used by PaymentClient, (2) retrieve the service name from the configuration used by PaymentServer, and (3) compare the retrieved names. The first and the second actions are executed in parallel, and the third one is executed after their termination.

Suppose the child autonomic manager associated with the third action determines that the DifferentNameSpaces diagnosis is the root-cause due to the fact that the retrieved service names are not equal. When the DifferentNameSpaces diagnosis is reported to the parent (current autonomic manager), the diagnoser of the parent maps the DifferentNameSpaces diagnosis into the NamingConflict symptom, and concludes the occurrence of NamingConflict.

4.5 Planner

The main responsibility of Planner is to select remedy plans from a pre-defined set of plans, specified by domain experts, based on the information contained in the relevant diagnoses. This section describes the entities related with the planner. Section 4.8.4 explains the information flow entity Plan and its internal structure.

Entities Related to Planner

Figure 4.8 shows the entities and relationships needed to select a plan. A Planner utilises one or more PlanSelectionRules to select suitable remedy plans based on the diagnoses determined. A plan consists of a collection of actions grouped by an ActionsConstruct. Planner’s activities are affected by the policy rules PlannerStrategicRules. An example of a policy rule is: ‘if there is more than one remedy plan then select the remedy plan with the highest weight value.’

Planner’s Process

The planning process model describes the working of Planner, shown in Figure 4.9. After the diagnostic process determines one or more diagnoses for the root-cause
of the current system malfunctioning, the planner is activated by the autonomic manager. The planner first executes its policy rules. It then selects one or more remedy plans, based on the given diagnoses, by executing its PlanSelectionRules.

**Figure 4.9:** Planning process model.

### Planner’s Role in the Scenario

When the autonomic manager, managing the execution of the *Payment* use-case in the failure scenario, notices that the diagnoser has inferred one or more diagnoses, it requests the Planner to select a remedy plan. As an example, suppose the determined diagnosis is *NameNotRegistered*. Based on this diagnosis, the remedy plan *ResolveNameConflict* executes two actions in sequence: (1) retrieve the service name from the configuration used by *PaymentServer*, and (2) modify the service name in the configuration used by *PaymentClient*. 
4.6 Plan Translator

The main responsibility of PlanTranslator is to map a specific plan onto effectors. Each ActionsConstruct has its own semantics. Actions are grouped in different ways such as sequentially and parallelly (see Section 4.8.4 for more details). The plan translator implements each ActionsConstruct as a method containing code to invoke the related effectors. This section describes the entities related with the plan translator. Section 4.8.5 explains the information flow entity, Effector.

Entities Related to Translator

Figure 4.10 shows entities and relationships that translate a plan. A PlanTranslator utilises one or more PlanTranslationRules that map a Plan onto one or more Effectors. An Effector is a piece of software instrumented in the code of a managed use-case, and listens to the instructions coming from its autonomic manager.

![Figure 4.10: The ER diagram depicts the relationships of a plan translator with its plan translation rules and policy rules. Furthermore, the diagram depicts how the plan translation rule uses a plan to translate it to an effector.](image)

PlanTranslator’s activities are affected by a set of policy rules represented as TranslatorStrategicRules.

Translator’s Process

The translation process model describes the working of PlanTranslator, shown in Figure 4.11. The plan translator is activated by the autonomic manager after the planning process selects a number of remedy plans to be translated. The plan translator first executes its policy rules. Thereafter, it attempts to map a remedy plan into adaptation instructions carried out by Effectors.

Initially, an Effector is inactive. When a plan translator translates a remedy plan into Effector, its autonomic manager sends a message to the Effector to activate it. When the thread of execution of the managed use-case reaches the point where the active effector is instrumented, the adaptation code is executed.
4.7 Autonomic Manager

The heart of the management model is the **AutonomicManager**. The main responsibility of an autonomic manager is to manage the execution of a use-case, and communicate the result of its management process to other autonomic managers. As mentioned before, the autonomic manager delegates its management tasks to its main entities analyser, diagnoser, planner, and plan translator, and coordinates their activities.

Policy rules (**ManagerStrategicRules**) influence the behaviour of an **AutonomicManager** (i.e., to make it customisable). When the autonomic manager starts, it first executes all of these rules. For example, policy rules are used to customise the **control flow** between activities of the autonomic manager’s main entities, or set the **threshold value** of a certain parameter. These rules are specified by domain experts (e.g., system administrators).

---

Translator’s Role in the Scenario

To heal the abnormal behaviour mentioned in the failure scenario, the plan translator acts as follows. Each adaptive action is translated by the plan translator into appropriate effectors. For instance, the second action of the remedy plan can be translated into the **ModifyServiceName** effector. This effector, defined by the domain expert, is instrumented at the instrumentation point (after the assignment statement in which the service name is assigned to a variable that represents the service name in the system) that is found in the **PaymentClient** sub-system. The effector is by default non-active. To perform an adaptation action, an autonomic manager turns the effector into the active state. When the thread of execution reaches the instrumentation point, the code of the effector is executed to modify the value of the variable representing the service name. This means that the self-healing is realised after the first occurrence of the **ServiceRetrievalFailed** symptom and before the second occurrence of that symptom.

---

![Diagram](image-url)  
**Figure 4.11:** Translation process model.
The next section explains how an autonomic manager coordinates the working of its main entities, and how it communicates with other autonomic managers.

4.7.1 Autonomic Manager’s Process

The autonomic process model describes the dynamic behaviour and lifespan of AutonomicManager. As illustrated in Figure 4.12, when an AutonomicManager starts, it activates its rule engine to execute the policy (strategic) rules. After execution of the policy rules, an AutonomicManager starts two parallel activities: waiting for sensor values coming from the managed use-case (WaitingForSensors state), and waiting for the autonomic process results coming from its children (WaitingForChildResult state).

When an AutonomicManager is in WaitingForSensors state, it receives information from various types of sensors. Sensors are instrumented in the code of a managed use-case and they can be triggered during the execution of the managed use-case. An AutonomicManager obtains information from JobStartSensor and JobEndSensor that have been instrumented at the beginning and at the end of the code of a managed use-case. They indicate that a managed use-case has started or the execution of the managed use-case has finished, respectively. AutonomicManager also obtains information from the sensors that have been instrumented in the content (core code) of the managed use-case. These are StateSensors or EventSensors that send information about the state changes and event occurrences to AutonomicManager. Based on the received information from a sensor, one of the following three activities is performed:

1. If AutonomicManager receives information from a JobStartSensor, it allocates resources with which to manage a running use-case, and activates its immediate children. After that, it returns to the WaitingForSensors state.

2. If AutonomicManager receives information from a state or event sensor instrumented in the content of the managed use-case, it activates the analysis process to ensure the occurrence of a symptom. If no symptoms can be inferred, AutonomicManager goes back to the WaitingForSensors state. If Analyser determines the occurrence of an abnormal behaviour, AutonomicManager activates the diagnostic process to find the root-cause of the symptom occurrence. If Diagnoser is not able to declare the reason for the abnormal behaviour, AutonomicManager does not take any action and returns to the WaitingForSensors state (i.e., AutonomicManager gives up and it notifies the system administrator about the current situation). Otherwise, AutonomicManager activates the planning process to select one or more remedy plans based on the inferred diagnoses. If there are no remedy plans available then AutonomicManager goes back to the WaitingForSensors state. Otherwise, it activates the plan translation process to map the selected remedy plans to appropriate effectors. After the translation process finishes its task, AutonomicManager goes back to the WaitingForSensors state no matter whether the translation process has translated a remedy plan or not.
3. If `AutonomicManager` receives information from a `JobEndSensor`, it deallocates the resources and deactivates its children. Subsequently, it sends its `AutonomicProcessResult` to its parent(s).

If `AutonomicManager` is in the `WaitingForChildResult` state and one of the children sends its autonomic process result, then the parent `AutonomicManager` activates the diagnostic process. At the same time, `AutonomicManager` returns to the same state to wait for the autonomic process results of its other children. After that, the autonomic process continues in the same way as explained above.
4.7.2 Relationship between Autonomic Managers

How the relationship among autonomic managers is determined is an essential issue in a self-managed system. In this thesis, the referential relationship among use-cases are used to this purpose. The referential relationship (both horizontal and vertical) among use-cases at different levels provides a corresponding binding among their autonomic managers. This binding is a parent-child relationship. Just before a child use-case is activated, the parent autonomic manager activates the appropriate child autonomic manager to control the execution of the activated child use-case. After the execution of the child use-case terminates (normally or abnormally), the child autonomic manager informs the parent about its management issues (such as problem determination, diagnosis, remedy plans).

Figure 4.13: Various use-case levels and their relationships including horizontal references, downward and upward vertical references. Moreover, the relationships of the autonomic managers, associated with each behavioural element (use-case), have been illustrated.

Figure 4.13 shows the different use-case levels containing behavioural elements (use-cases) including their horizontal and vertical references. Sections 3.6.1 and 3.6.2 explain the horizontal and vertical references in the figure, respectively. The pentagon above each behavioural element represents an autonomic manager and the arrows between autonomic managers illustrate their relationships.

For the sake of clarity, the relationships of the three use-cases BE1 and BE2 at system level, and BE3 at runnable level and their associated autonomic managers AM1, AM2 and AM3 are re-examined in a separate figure (see Figure 4.14). The com-
Figure 4.14: Behavioural elements (i.e., use-cases) and their autonomic managers. BE1 references (i.e., calls) BE2 and BE3. AM1 communicates with both AM2 and AM3 to coordinate the management of the complex use-case BE1.

Complex use-case BE1 references BE2 (horizontal reference) and BE3 (vertical reference) in order to compute and construct its result. The autonomic managers AM1, AM2 and AM3, on the one hand, control the execution of BE1, BE2 and BE3 through a feedback loop. On the other hand, AM2 and AM3 communicate with AM1 to inform it (parent) about any management issues that have arisen during the execution of BE2 and BE3.

Figure 4.15: The ER diagram depicts the relationships of the autonomic manager with its parents and children. It also depicts the unit of communication (AutonomicProcessResult) between autonomic managers.

Based on the description above, Figure 4.15 depicts entities and relationships representing the relationship between autonomic managers. The hasChild relation for an AutonomicManager is introduced to relate autonomic managers with each other based on the relationships between their managed use-cases. The relation refers to a set of AutonomicManagers that are children of an autonomic manager. Each autonomic manager can have zero or more child autonomic managers, indicating that its managed use-case can reference multiple other use-cases. The relation also refers to a set of AutonomicManagers that are parents of an autonomic manager. Each autonomic manager can have zero or more parent auto-
nomic managers, indicating that its managed use-case can be shared by multiple other use-cases.

A child autonomic manager communicates the result of its process by passing an `AutonomicProcessResult` to its parent. For a parent autonomic manager two things about its child are important: (1) the opinion of its child regarding the root-cause of a problem, and (2) the remedy actions performed by its child. To this purpose, `AutonomicProcessResult` contains the `determinedDiagnosis` and `executedRemedyPlan` relationships to inform the parent about how the child use-case has executed. The entities representing the determined diagnosis (Hypothesis) and the executed remedy plan (Plan) are explained in more detail in Sections 4.8.3 and 4.8.4, respectively. Note that although in Figure 4.14, the reference from BE1 to BE2 is a horizontal one and the reference from BE1 to BE3 is a vertical one, both AM2 and AM3 use the same entity (AutonomicProcessResult) to communicate the result of their autonomic process with their parent (AM1).

4.8 Information Flow Entities

The autonomic manager’s main entities communicate with the managed use-case through `Sensors` and `Effectors`. The main entities communicate with each other with `Symptoms`, `Hypotheses`, and `Plans`. These entities are populated with appropriate information either during the execution of a managed use-case or by activation of a main entity of the autonomic manager. They are called information flow entities. These entities are stored in the knowledge-base shared by all main entities of the autonomic manager. This section explains the data structure of these entities and their relationships.

4.8.1 Sensors

`Sensors` acquire appropriate information during the execution of a managed use-case. Note that a managed use-case is implemented by a software program, and a `Sensor` is a piece of software that is instrumented around the code implementing a use-case step of that managed use-case. Sensors are used by the autonomic manager and the analyser to provide knowledge about the current state of a managed use-case. Knowledge is captured during execution of a use-case by observing state changes, event occurrences, interactions with the environment, and communications between use-cases. These observations are passed on to the analyser. Each sensor contains not only information about the value of an item to be monitored but also meta-information about the domain of these values, the type of monitored item, and the type of instrumentation placed in the code. Figure 4.16 depicts the relationships of a sensor and its attribute.

**Sensor - Monitored Item**

A `Sensor` monitors state changes and event occurrences as execution of a use-case step changes the state of a managed use-case. The variables in the code of
Figure 4.16: The ER diagram depicts the relationships of a sensor, and its attribute. It depicts how a sensor is instrumented in a managed use-case (InstrumentationType), what it monitors (MonitoredItem), and what the type of its observed value (ValueDomain) is.

A managed use-case are represented by State, and the different types of events that may be identified during code execution are represented by Event. State and Event are sub-entities of MonitoredItem. They are part of the behavioural model, and described in more detail in Sections 5.1.3 and 5.1.4. To classify observations, based on MonitoredItem, two types of sensors are distinguished: StateSensor and EventSensor. A StateSensor observes changes regarding a state that were caused by execution of a use-case step. An EventSensor observes occurrence of an event caused either by execution of a use-case step or by an environmental factor. Both sensor types are further subdivided into a hierarchy of sensor types that corresponds to the hierarchy of states and events specified in the behavioural model.

Sensor - Instrumentation Type

A Sensor is a piece of software code that is woven (instrumented) into the code of a managed use-case at a pre-defined point. The software code of a sensor is invoked when the thread of execution of a managed use-case reaches that pre-defined point. The InstrumentationType, shown in Figure 4.16, represents a point in the code of the managed use-case where a Sensor is instrumented. Various instrumentation points (InstrumentationTypes) are identified (see Figure 4.17) that are explained below: MethodCall, MethodExecution, HandlerExecution, FieldAccess, and LineNumber.

A MethodCall represents a point immediately before the invocation of another use-case or after returning. A sensor, instrumented at this point, monitors the communication between the managed use-case and other use-cases. The input values of an invoked use-case are checked before invocation. After invocation, the result of the invoked use-case is examined.

A MethodExecution represents a point at the beginning or end of a managed use-case. By instrumenting a sensor at the beginning, the expected input values of a managed use-case are checked. The instrumented sensor at the end of the managed use-case monitors the result of the execution of the managed use-case.
MethodExecution is also used to instrument sensors that indicate the lifespan of a managed use-case, and sensors that capture unexpected events (i.e., events for which a managed use-case does not include code to handle).

A HandlerExecution represents a point at the beginning or end of the execution of a piece of code of a managed use-case that handles exceptional situations that are raised either by execution of a use-case step or by external factors. The goal is to monitor the behaviour of the handler included in the managed use-case.

A FieldAccess represents a point immediately before or after a state is read or written. By instrumenting a sensor at this point, the expected state changes are checked. These changes are the result of execution of a use-case step.

Once an instrumentation type is specified for a Sensor, the self-management framework proposed in this thesis (see Chapter 8) can automatically find the corresponding instrumentation points in the software program (code of a managed use-case), and instrument a Sensor in a program. As basically a Sensor monitors a use-case step and a use-case step can be implemented by a group of statements within the program, there is a possibility to instrument a sensor before or after a group of statements. A LineNumber is introduced to provide an opportunity to automatically instrument before or after a specified line of code.\footnote{This does not necessarily imply that the source code of a managed system should be available for the autonomic management. If a software program is compiled with the debug option, the compiler puts line number information (the line number of each statement) in the compiled file.}

![Figure 4.17: Various places in the managed use-case where sensors can be instrumented.](image)

**Figure 4.17:** Various places in the managed use-case where sensors can be instrumented.

### Sensor - Value Domain

After a Sensor is associated with a MonitoredItem and is instrumented in the code of a managed use-case, it passes observed values of MonitoredItems to Analyser. Each observed value is then compared with the expected value to derive the occurrence of a symptom. To compare two values, both have to belong to the same ValueDomains. A ValueDomain can be: BooleanType, IntegerType, DoubleType, DateType, TimeType, CharType, or StringType. These entities represent the most widely accepted primitive value types, inspired from XML Schema specification \cite{XML Schema}. ValueDomain is not limited to the data types mentioned. It can be extended to include new data types.

Chapter 3 stressed the importance of contextual knowledge in performing diagnos-
4.8 Information Flow Entities

tic tasks. A Sensor provides contextual knowledge as qualitative and quantitative temporal information, and spatial information regarding each observation. Qualitative temporal information refers to the chronological order of a managed use-case execution in a chain of use-cases, and the chronological order of a use-case step execution in the context of a managed use-case. Quantitative temporal information is represented by the timestamp attribute of Sensor, and refers to the actual timestamp of the observation. The spatial information about an observation is provided in the form of information concerning the structural element where the Sensor has been instrumented.

4.8.2 Symptoms

The value of a Symptom is determined by an analyser and is used by a diagnoser. A Symptom represents abnormal behaviour. An example symptom, that can be defined by a domain expert, is highAmountSymptom that occurs when the value of a transaction exceeds a certain threshold amount. A Symptom is either derived by one or more symptom occurrence rules that use the values of the relevant Sensors or by performing an inspective plan (see Figure 4.18). The arisen attribute of Symptom shows whether the Symptom has occurred or not. This attribute is of type Ternary that can have three values: unknown, pos, and neg. The first value indicates that it is not known whether a symptom has occurred or not. The second one indicates that a symptom has indeed occurred (positive), and the third one indicates that a symptom has not occurred (negative). Initially, all Symptoms are set to unknown.

![Figure 4.18: The ER diagram depicts the relationships of a symptom, and its attribute. Each Symptom is associated with one or more Sensors. A Symptom is determined either by the values of the relevant Sensors or by performing an inspective plan.](image)

The inspectivePlan relation of Symptom needs more explanation. While determining a diagnosis based on symptoms, it is required that the arisen attribute has the value pos or neg but not unknown. If the value is unknown then the diagnostic engine of the autonomic manager pro-actively performs the inspective plan to confirm occurrence or absence of the symptom.

Chapter 3 introduced use-case levels to structure a considerable number of complex use-cases of a managed system. Depending on the type of managed use-case (i.e., the use-case level it belongs), symptoms can be SystemLevelSYM, RunnableLevelSYM, ComponentLevelSYM, and ClassLevelSYM. The system level symptoms concern malfunctions experienced by end-users, the runnable level symptoms concern infrastructural malfunctions, the component level symptoms
concern malfunctions related to the behaviour of functional and technical components, and the class level symptoms concern malfunctions that are specific to classes and methods.

### 4.8.3 Hypotheses

The value of a Hypothesis is determined by a diagnoser and is used by a planner. A Hypothesis represents the possible root-cause of the occurrence of a symptom, and is specified by a domain expert. The diagnoser uses a pre-defined set of hypotheses to indicate some of those hypotheses as diagnoses. An example hypothesis might be directoryNotShared that describes the root-cause of the unsuccessfulReadAction symptom (not being able to read data from a shared file).

![Figure 4.19: The ER diagram depicts the relationships of a hypothesis, and its attributes. Attributes together with the relevantSymptom and subHypothesis relationships are used to reason whether a hypothesis can be marked as diagnosis. The other relationships provide information regarding how and where the abnormal behaviour has occurred.](image)

The primary goal of finding the root-cause of malfunctioning is to understand why and how a symptom occurs so that it can be prevented from recurring. Therefore, a diagnostic task should determine which part of a managed system caused the malfunctioning, and under which conditions malfunctioning occurred. Figure 4.19 shows all attributes and relationships of Hypothesis that are needed to express the mentioned goal.

The attributes of Hypothesis are used by the diagnoser, during the diagnostic process, to determine a diagnosis. The attributes focussed, assessed, and determined are of boolean type, and are initially set to false. They become true if the hypothesis is selected for validation, evaluated, and marked as diagnosis. The attribute validated is of type Ternary, and is initially set to unknown. It becomes pos or neg if the hypothesis is validated. The attribute weight is of a numeric type, and it is determined by domain experts to indicate the degree of possibility that the Hypothesis is the root-cause. The value of this attribute is used by the
4.8 Information Flow Entities
diagnosis determination rules to choose the hypothesis with the highest weight value.

The relationships relevantSymptom and subHypothesis play a role during diagnostic reasoning. When an abnormal behaviour (symptom) occurs, the hypothesis selection rules select hypotheses that can explain the root-cause of that symptom. The relevantSymptom relates Hypothesis with one or more Symptoms that can be explained by the Hypothesis.

To structure hypotheses and facilitate the selection process, they are organised as a hierarchy of hypotheses. The more generic hypotheses are placed at the top of the hierarchy. The relation between hypotheses in the hierarchy is expressed by subHypothesis that points to a child hypothesis. For example, the directoryNotReadable hypothesis can be considered as the sub-hypothesis of the fileSystemNotAvailable hypothesis. When both fileSystemNotAvailable and directoryNotReadable are selected, validated, and evaluated, the diagnoser can use the hypothesis hierarchy to determine more generic root-causes.

Section 3.2 explains why contextual information is important in performing diagnostic tasks. To provide this information, three relationships faultyElement, faultyJob, and faultyTask are defined for each Hypothesis. The first one relates a Hypothesis to a ManagedStructuralElement\(^6\) that specifies the portion of the system code that caused the problem. The second one relates a Hypothesis to a Job that represents the managed use-case. The third one relates a Hypothesis to a Task\(^7\) representing a use-case step that caused the malfunctioning.

Hypotheses can be SystemLevelHYP, RunnableLevelHYP, ComponentLevelHYP, and ClassLevelHYP. This means that each hypothesis type describes the root-cause of its corresponding symptom type (e.g., a RunnableLevelHYP describes the root-cause of a RunnableLevelSYM).

4.8.4 Plans
The value of a Plan is determined by a planner and is used by an autonomic manager, diagnoser, and plan translator. A Plan is defined as a collection of actions (each represented by a Job), and the way (e.g., in sequence or in parallel) these actions are executed. The Jobs comprising a Plan are not part of a managed system, and are started by an AutonomicManager. Each such Job is managed by its own AutonomicManager specified to be the child of the current AutonomicManager.

Figure 4.20 shows the different types of plans distinguished in the management model. An InitialPlan is a collection of actions that are started by an autonomic manager immediately after it starts. An example of an initial plan are jobs that are required for gathering environmental information, or required for starting timers to perform periodic tasks.

---

\(^6\)The ManagedStructuralElement entity is part of the system structural model, and is described in Section 5.2.

\(^7\)The Task entity is part of the system behavioural model, and is described in Section 5.1.
A RemedyPlan is a collection of actions performed to overcome a certain malfunctioning within a managed use-case. RemedyPlans are further categorised into SelfConfiguringPlan, SelfHealingPlan, SelfOptimisingPlan, and SelfProtectingPlan based on the most widely accepted autonomic properties (see Chapter 1).

Finally, an InspectivePlan is a collection of actions associated with a symptom. The inspective actions are performed by diagnosers in order to help the diagnostic engine to make correct decisions and to properly validate a hypothesis.

Plans are categorised into SystemLevelPLN, RunnableLevelPLN, ComponentLevelPLN, and ClassLevelPLN. This categorisation is also based on the four use-case levels, similar to categorising symptoms and hypotheses into the four groups. This way, the use-case level specific symptoms, hypotheses, and plans can be related with each other (e.g., a RunnableLevelSYM is used by the diagnoser to determine a RunnableLevelHYP that is used by the planner to select a RunnableLevelPLN). The assumption is that a managed use-case belonging to a specific level shows the level-related symptoms caused by the level-related root-causes, and requires level-related remedy plans.

**Actions Constructs**

An ActionsConstruct is a control construct that bundles various actions of a Plan, and specifies how these actions are executed. Domain experts can use ActionsConstructs to compose different actions that are executed by an autonomic manager. Each action is defined to be either a Job or, again, an ActionsConstruct. For instance, consider a plan, called myParPlan, that consists of two sets of actions ($s_1 = \{a_1\}$ and $s_2 = \{a_2, a_3\}$) to be executed in parallel. Moreover, actions in $s_2$ should be executed in sequence. To specify a plan, first the three actions $a_1$, $a_2$ and $a_3$ are defined as Jobs. Then the ActionsConstruct $ac_{seq}$ is defined to bundle the actions $a_2$ and $a_3$. Last, the ActionsConstruct $ac_{par}$ is defined such that it bundles the actions $a_1$ and the ActionsConstruct $ac_{seq}$.
Different types of control constructs are identified (see Figure 4.21). The following describes each of these constructs.

The **AnyOrder** construct bundles a number of actions that are to be executed in some unspecified order. **Choice** groups a number of actions of which only one is selected for execution. Domain experts specify actions and a control construct for these actions, and autonomic managers execute the construct. For example, when an autonomic manager interprets **AnyOrder**, it executes all actions contained in **AnyOrder** in an unspecified order, or when an autonomic manager interprets **Choice**, it executes only one of the actions contained in **Choice**. It is up to the autonomic manager to choose one.

**IfThenElse** allows conditional execution of two collections of actions. **IfThenElse** has three attributes: **ifCondition**, **then** and **else**. The **ifCondition** attribute of this construct points to a set of logical expressions (rules), and the **then** and **else** attributes point to a set of **ActionsConstructs**.

The **RepeatUntil** and **RepeatWhile** constructs allow that a collection of actions are repeatedly executed until a certain condition becomes true (for the first construct) or false (for the second construct). The first construct has the attribute **untilCondition**, and the second one has the attribute **whileCondition**. The **untilCondition** and **whileCondition** attributes of these constructs refer to a set of logical expressions.

**Sequence** is the opposite of **AnyOrder**. It enforces execution of a group of actions in a specified order. The **Parallel** construct allows actions in a specified collection to be executed in parallel. All control constructs, except for **IfThenElse**, have the attribute **elements** that refers to a set of **ActionsConstructs**.

Finally, the **JobInvocation** construct contains one action. The important point is that this action refers to a **Job** (a basic element of the behavioural model). This job is called an **administrative job**. An administrative job is specified in the same way as a regular job. It is invoked by an autonomic manager, and it is associated with its own autonomic manager. The autonomic manager of

---

8 The semantics of these control constructs is similar to the semantics of control constructs used in the OWL-S specification [130] for the description of composite processes.
an administrative job becomes the child of the autonomic manager that invokes that administrative job. This ensures that all management tasks of autonomic managers themselves become the subject of self-management.

### 4.8.5 Effectors

An **Effector** is populated with appropriate information by a plan translator. It represents a piece of software code that is instrumented, similar to a **Sensor**, around the code implementing a use-case step of the managed use-case. The execution of the effector code adapts the current behaviour of the managed use-case.

![ER Diagram](image)

**Figure 4.22:** The ER diagram depicts the relationships of an effector and its attribute. Each Effector specifies how it is instrumented in the managed use-case (**InstrumentationType**), what it adapts (**MonitoredItem**), and what the type of its adaptation value (**ValueDomain**) is.

Figure 4.22 illustrates the relationships of Effector and its attribute. The entities MonitoredItem, InstrumentationType, and ValueDomain were already explained in Section 4.8.1. An Effector adapts the value of a State or the code of the handler of an Event when its isActive attribute is set to true by the plan translator.

### 4.9 Related Work

A number of design choices, concerning the management model of the self-management framework presented in this thesis, have been made. The high-level design choices involve the architecture of an autonomic manager, and the analysis, diagnosis, and adaptation techniques used by the autonomic manager’s main entities. In this section, some related works regarding these design choices are reviewed.

**Autonomic Manager’s Architecture**

The internal architecture of an autonomic manager in a self-management framework determines to a large extent the maintainability, transparency, extensibility,
and adaptability of the framework. The study of the research works shows that the degree of attention of researchers to the internal architecture of an autonomic manager depends on the generality of their proposed self-management framework.

The following reviews the internal architecture of an autonomic manager in two types of generic self-management frameworks: autonomic programming frameworks [20, 125] and autonomic application frameworks [33, 39].

Accord [125] is an example of an autonomic programming framework. Autonomic components are the building blocks in Accord. An autonomic component is a self-contained and modular component with specified interfaces, rules, constraints, and mechanisms for the self-management. An autonomic component mainly consists of a rule agent, that is responsible for periodically querying the state of the component and controlling the firing and execution of the rules, and a computational component, that is the traditional component and is responsible for performing computational tasks. A rule agent in Accord resembles an autonomic manager. The internal architecture of a rule agent is not known. It seems that Accord makes no distinction between different functionalities within an autonomic manager. The managed resource in Accord is a software component. Another example of an autonomic programming framework is ABLE [20]. ABLE is introduced as a platform for constructing autonomic agents. The number of components in an autonomic agent in ABLE is extensive. Three behavioural components are used to implement the basic (reflexive), the complex (reactive), and the more complex (learned) behaviours of an agent. An executive component is responsible for making decisions based on two models: agent’s self-model and agent’s world model. An action planner component creates sequences of actions necessary to achieve a goal determined by the executive component. The managed resource in ABLE is a software agent.

An example of an autonomic application framework is JAGR [33]. JAGR is a self-recovering framework that integrates its self-recovery components into the opensource J2EE application server JBoss [100]. An autonomic manager in JAGR contains the following components: monitoring, recovery management, recovery agent, fault injector and stall proxy. Its monitoring component tracks Java level exceptions thrown by application and platform components, and reports the error and the offending component to the recovery manager. Based on the monitoring information and path-based failure analysis, it detects unexpected component behavior. To reduce recovery time of the whole system, it reboots only those components that have been affected by the failure through inspecting the component dependency graph. The managed resource in JAGR is an Enterprise Java Bean (EJB) component. Rainbow [39] is another example. Rainbow uses an abstract architectural model of a managed system to monitor and evaluate the system’s runtime properties, and adapt the system. An autonomic manager in Rainbow contains four main modules: model manager, constraint evaluator, adaptation engine, and adaptation executor. These modules are responsible for maintaining an annotated architectural model (consisting of components and connectors annotated with desired properties and constraints) of a managed system, evaluating the model for constraint violation, and performing adaptations on the running
The Management Model

system. The managed resource in Rainbow is an architectural element (a system’s component or connector).

Note that some researchers [10, 14, 96] either skip to implement one or more main functionalities of an autonomic manager or do not make a clear distinction between them. For example, in the knowledge-based framework for multimedia adaptation [96], there are no implementations of the analysis and diagnosis functionalities. An autonomic manager in this framework contains only two functionalities: adaptation plan and adaptation service. Other examples are the works of Ardissono et al. [10] and Baresi et al. [14], who applied the general principles of the autonomic computing to the fault diagnosis of Web Services and monitoring of Web Service compositions, respectively. The only functionality implemented in the work of Ardissono et al. is the diagnosis. In the work of Baresi et al., the analysis and diagnosis functionalities are highly integrated in one component containing monitoring rules.

In the self-management framework proposed in this thesis, similar to ABLE and Rainbow, the complex management task is decomposed into a number of less complex sub-tasks that are performed by well-defined separate software entities (i.e., analyser, diagnoser, planner, and plan translator). This separation of concerns results in a more maintainable and adaptable autonomic manager. The managed resource in our framework is a system behaviour (specified as a use-case). Despite of that, the framework takes into account the system’s components and connectors, similar to Rainbow and JAGR.

Analysis Techniques

The analysis process is responsible for inspecting the collected data, that represent the current status of a managed system, to determine any abnormality. The number of analysis techniques used by different researchers to realise self-management are moderately large. Dependency analysis [105, 107], program analysis [12, 67, 141, 196], path-based analysis [34, 35], and learning-based analysis [4, 59, 186] are examples of analysis techniques.

To perform root-cause analysis of system malfunctions, some researchers propose the use of **dependency models** that describe dependencies between system components. A dependency model is usually represented as a directed acyclic graph in which nodes represent system components and weighted edges represent dependencies. Keller et al. [107] propose two different types of dependency models: functional and structural. Dependency models can be constructed either manually or automatically. Recently, the automatic identification of components dependencies has received increased attention. Techniques have been proposed for this purpose are: instrumenting within the application and service components to capture invocations from one component to the other [107], instrumenting the communication protocol stack to intercept the communication between components [138], using information stored in system configuration repositories [105], and active perturbation of components by injecting faults and observing the behaviour of the
4.9 Related Work

Program analysis is the process of automatically analysing a computer program in order to predict the program’s behaviour at runtime. The result of program analysis is used to diagnose the root-cause of software faults. Program analysis offers a number of techniques such as data flow analysis, control flow analysis, model checking, program slicing [141], etc. Program Dependence Graph (PDG) [67] has been proven to be the preferred data structure for representing certain control or data dependencies between program statements. However, Baah et. al [12] argue that PDGs are not able to model the statistical (uncertain) dependencies between program elements. Therefore, they propose a new graph, called Probabilistic Program Dependence Graph (PPDG), that facilitates probabilistic reasoning about program behaviours for fault localisation and fault comprehension. Zoeteweij et al. [196] propose an automated debugging technique through program analysis based on program spectra for locating software faults. They define a program spectrum as a collection of data (a vector), collected at runtime, that characterise a specific behavior of the program and indicate which parts of the program were active during various executions of that program. A very simple program spectrum is a block count spectrum telling how often each block of code is executed during a program run. For fault diagnosis, they constitute a binary matrix whose columns correspond to different parts of the program (program blocks) and rows correspond to different program runs. There is also an error column whose elements indicate whether an error occurred during a specific program run. In this way, they try to find correlations between various program spectra and the errors detected in the different program runs.

The path-based root-cause analysis pays more attention to behaviour of a system rather than the structure of a system’s components. In the path-based approach, the paths that requests follow as they move through the system are recorded [34, 35]. A request is defined as any external entity that asks the system to perform some action, and a path is defined as a collection of connected resources associated with servicing a request. The target system is modelled as a collection of paths. Paths may have dependencies through shared resources. For diagnosing failures, large volumes of requests are statistically analysed to identify significant deviations from normal behaviour.

Many researchers propose utilising various Machine Learning (ML) methods to automatically create and continuously adapt the required knowledge for analysing a system behaviour. For example, Wildstrom et al. [186] propose an ML based approach whereby the autonomic manager learns to predict the change in resource requirements of the system. The training data (to train the performance prediction models) consists of kernel thread counts, memory utilisation, paging events, system events, and CPU usage for a range of configurations and workloads. The knowledge-base containing the trained models is used to estimate the expected gain or loss of a resource reallocation. Al-Nashif et al. [4] use ML for their autonomic multi-level network intrusion detection system. They use an unsupervised learning algorithm to identify changes in network operations at three different levels: network flow, network protocol, and network payload. If these changes show...
a discrepancy with the current baseline models (that have been produced by the learning algorithms and are available in the knowledge-base) of normal network operations then an adaptive supervised learning is used to re-train the errant system in real time. Duan and Babu [59] use ML for failure diagnosis. They collect monitoring data from failure states of the system and annotate the data with information about the type and cause of failure. When the system experiences a failure, they try to determine whether the failure is the same as a previously diagnosed failure. They view this as a multi-class classification task, and therefore, they use a decision tree to train a classifier from the current set of annotated failure instances. The decision tree and the different failure classes are stored in the knowledge-base. If the cause of failure instances is unknown, they use sophisticated unsupervised ML methods to group failure instances into clusters that contain instances of the same failure type with high probability.

In this thesis, different dependency models are combined to perform analysis. In our primary dependency model, nodes do not represent system components but system behaviours (use-cases), and edges represent both compositional and usage dependencies (see Section 3.6). In our secondary dependency model, nodes represent system components and edges represent compositional dependencies. These models are more or less comparable with the functional and structural dependency models of Keller et al. [107]. Furthermore, in our third dependency model, nodes represent both system behaviours and system components, and edges represent usage dependencies between system behaviours and system components. A use-case in our approach can be considered as a combination of a request and a path in a path-based approach [34]. The difference is that a path only contains information about a system’s components (that are responsible for executing the relevant request), and it does not contain information about a system’s states and events (that change/occur during execution of the relevant request by the system). Similar to the work of Duan and Babu [59], the analyser in our approach collects monitoring data from failure states and events of a system, and annotates the data with information about the type of the failure states and events. However, our approach uses logical rules, instead of ML methods, to perform analysis (determine a symptom), based on the monitoring data and the dependency models.

Diagnosis Techniques

The diagnosis process is responsible for finding the root-cause of any abnormality determined by the analysis process. Examples of diagnosis techniques used for self-management purposes are: model-based diagnosis [10, 11, 150], case-based diagnosis [137, 184], and rule-based diagnosis [50, 54, 80, 110].

The term Model-Based Diagnosis (MBD) refers to a diagnostic approach where a system to be diagnosed is modelled as a set of components and their interactions. In MDB, a diagnosis is considered as a set of assignments of behaviour modes (faulty or correct mode) to a set of components for a given set of observations (runtime values of the component’s variables). There are two MBD
4.9 Related Work

approaches: consistency-based or abductive. According to the first approach, the component’s expected correct behaviour is modelled. If a component shows an abnormal behaviour then it is concluded that the observation is not consistent with the model any more. This approach reasons from causes to effects. According to the second approach, the component’s faulty behaviour is modelled. The observation of an abnormal behaviour implies the correctness of the predicted faulty behaviour. This approach reasons from effects to causes. Originally, MBD has been developed for determining faults on physical systems (e.g. electronic circuits). Recently, they have also been applied to find faults in component-oriented software programs [150]. Ardissono et al. [10, 11] propose a model-based approach to the fault diagnosis of Web Services. In their approach, each activity (method) of a Web Service corresponds to a component in MDB terminology. The behaviour of a Web Service is modelled as a correlation between input and output parameters of the different activities. The model is a set of ok (representing normal behaviour) and ab (representing abnormal behaviour) values that are assigned to the input and output parameters of all activities. The runtime values of the parameters (observations) are provided by loggers embedded in the code of each activity. The diagnoser is triggered if there is a conflict between the model and the observations.

The main element in Case Based Diagnosis (CBD) is a knowledge-base of past situations (cases) that are reused in solving present problems. A case is a set of meaningful features of a specific problem together with the applied solution to that problem. A typical CBD process includes four phases: retrieve, reuse, revise, and retain. To find a suitable solution for a new problem, the features of the problem are matched against cases in the knowledge-base and one or more similar cases are retrieved. After that, the associated solution of the retrieved cases are reused and tested. If necessary, the retrieved solution is revised producing a new case that is retained in the knowledge-base. By retaining more and more representative examples in the knowledge-base, it becomes easier to find a suitable solution to a new problem [184]. Montani and Anglano [137] provide an excellent reference to the work of researchers who apply the principles of CBD to failure diagnosis and remediation in software systems.

In Rule Based Diagnosis (RBD), the empirical knowledge of domain experts are elicited and implemented in the form of rules. Each rule consists of one or more conditional sentences relating statements of facts with one another. The diagnosis is realised by using a causal model and filtering out those rules matching the system observation. In spite of the fact that the knowledge elicitation is a difficult process, some researchers [31, 119] still prefer RBD over other diagnostic approaches. They argue that the rule inference engines are more flexible than other reasoning engines, and use reasoning which more closely resemble human reasoning. De Paola et al. [50] apply RBD to network management. In their distributed multi-agent architecture for network management, a rule-based reasoner is not only capable of diagnosis but also able to trigger monitoring and repair network anomalies. For diagnosing failures in large-scale network protocols, Khanna et al. [110] propose a monitoring architecture through which the message exchanges between the protocol entities are observed. Then, the observed data are stored in a causal
The Management Model

At runtime, a rule-based diagnoser uses this graph together with a rule base of allowed state transition paths to diagnose the failures. One of important shortcomings of RBD is its inability to deal with unknown situations. If the observation does not exactly match the condition of a rule in the rule base, no diagnosis is performed by the rule inference engine. To overcome this shortcoming of RBD, Hanemann [80] combines RBD with CBD. In his proposed service-oriented event correlation framework, both diagnosers run in parallel. When an event cannot be matched to any rule in the rule base, the case-based diagnoser is triggered to match the current event to prior cases. If the previous solution associated with a prior case is not adequate, the system administrator is requested to adapt the prior solution or determine a new solution (root-cause of the case). Then, the current event together with its solution is used to automatically generate a new rule and update the rule base with that new rule to cope with similar events in the future.

A variant of rule-based reasoning that has attracted interest of many researchers is fuzzy rule-based reasoning that uses fuzzy set theory [193] to deal with approximate rather than crisp (precise) reasoning. A simple fuzzy rule is usually expressed in the form of ‘IF variable IS property THEN action’ (e.g. ‘IF temperature IS very hot THEN start fan’) [60, 75]. Also, the usage of fuzzy rules in control systems (i.e. systems containing closed-loop feedback) is an active area of research [148]. For example, Diao et al. from IBM T.J. Watson Research Center [54] use a fuzzy controller to self-optimize the response times of the Apache Web Server [8].

Although in our approach, similar to MDB, a set of components and their interactions, a set of observations, and a set of behaviour modes (symptoms) are defined, we do not use the classic MDB. To our knowledge, the MDB does not take into account that the occurrence of a behaviour mode can be unknown, and it should be determined during the diagnosis process. This aspect should be taken into account in a fault diagnosis approach for finding root-cause of failures in software programs. This thesis solves the issue of unknown behaviour modes by combining the ternary logic based diagnostic model of Brazier et al. [24] with the description logic. To realize more effective diagnosis, combining RDB with CBD would be a reasonable option, as suggested by Hanemann [80].

Adaptation Techniques

One of the biggest challenges of the autonomic computing paradigm is system adaptation. An adaptable system should be able to dynamically react and modify its behaviour in response to changes in its execution environment. Different approaches are taken by various researchers to realize software adaptability [28, 140, 146]. Here, a very limited number of adaptation approaches (dynamic reconfiguration [19, 37, 38] and AOP-based adaptation) are reviewed.

Dynamic reconfiguration is a software mechanism that allows resources to be added or removed from the system without bringing the system down. Dynamic reconfiguration requires minimal execution disruption and some well-defined consistency preserving of the system. Bidan et al. [19] apply dynamic reconfiguration
in the CORBA framework. A CORBA object is brought to life when a client application requests an operation on that object. The request is sent to the CORBA object using the Remote Procedure Call (RPC) [174] mechanism. To perform reconfiguration and, at the same time, guarantee the consistency constraint, Bidan et al. state that a reconfiguration action should not leave initiated RPCs pending. Therefore, a remote call whose target is a CORBA object that is going to be reconfigured is blocked. The remote call remains blocked for the duration of the requested reconfiguration action. After that, the request is routed to the adapted (reconfigured) CORBA object. Xuejun [37, 38] extends Java RMI to preserve the consistency of distributed Java systems when reconfiguring a component. His extended RMI (XRMI) includes a software layer on top of Java RMI. The additional layer contains a virtual stub and a configuration management agent (CMA). A virtual stub is an object that is used by a client component to communicate with a server component. CMA is responsible for the reconfiguration of all server components. In addition, it maintains a list of pairs, each pair consisting of a virtual stub and its server component. When a server component needs to be reconfigured, CMA asks the virtual stub of the server component to block any new invocations to its related server component. After the reconfiguration, the virtual stub is signaled by CMA to update the real reference to the server component and to resume the blocked communication.

Aspect-Oriented Programming (AOP) is an appropriate technique for software adaptation that makes use of aspects to facilitate the dynamic adaptation of components and services transparently and in a non-intrusive way. Multiple behaviours of a component (or a service) are expressed in separate aspects. The system adaptation is realised by replacing the current aspect with a new one depending on the changes in the environment. Different researchers utilise AOP for adaptation purposes in different ways. For example, for runtime adaptation in a Service Oriented Architecture (SOA) environment, Irmert et al. [94] integrates AOP with the OSGi Service Platform [5] that supports hot-deployment. A deployable unit in a OSGi platform is called an application bundle. Several application bundles can co-exist inside the platform that runs on a single Java Virtual Machine (JVM). The platform provides various API's for installing, starting, stopping and de-installing application bundles without restarting the platform. In their approach, AOP aspects implementing different behaviours of a service are deployed as OSGi application bundles. The deployed aspects are maintained and managed by the aspect manager. When an adapted behaviour of a service should replace the current behaviour, the aspect manager utilises the hot-deployment capability of the OSGi platform to stop/de-install the current aspect and install/start the new one representing the adapted behaviour. Another example of AOP-based adaptation is the TRAP [159] framework. TRAP uses AOP to adapt existing object-oriented applications. It performs the adaptivity task at compile time and runtime. At compile time, for each base class to be adapted, the framework produces three classes: an aspect class, a wrapper class extending the base class, and a meta class. The generated aspect class contains an AOP advice causing the instantiation of the wrapper class instead of the base class. The generated meta class contains a list of delegate
objects, each providing specific implementation of a base class method. In other words, each delegate object implements an alternative behaviour of the same base class to be adapted. At runtime, using an interactive management console, users can register/unregister the delegate objects with the meta class for the purpose of adapting the behavior of the base class.

This thesis prefers the AOP-based techniques for software adaptation for the following reasons. First, in an AOP-based adaptation (in contrast to a dynamic reconfiguration approach), it is possible to adapt the behaviour of an existing component in a system, in addition to adding or removing a component. Second, adaptation in an AOP-based adaptation approach is more modular. Adaptations are expressed in separate aspects which are reusable programming constructs that can be carefully designed and programmed. In our approach, adaptations are considered as new behaviours. Similar to the existing system behaviours, these new behaviours are also represented as use-cases that are implemented as aspects in an AOP language. The meta-information about these aspects are included in the management model expressed in a declarative language.

### 4.10 Summary

This chapter explains the management model for distributed systems. The management model contains three category of entities: `AutonomicManager`, its main entities (`Analyser`, `Diagnoser`, `Planner`, and `PlanTranslator`), and information flow entities (`Sensor`, `Symptom`, `Hypothesis`, `Plan`, `Effector`). This chapter explains how the structured information (that is now in a format understandable for a computer) can be used to automate problem determination and repair actions, to heal software malfunctioning.
Chapter 5

The System Model

The previous chapter introduced a management model for distributed systems. This chapter introduces a model of the system to be managed, a distributed system. The model contains both behavioural and structural models. The behavioural model, presented in Section 5.1, describes the way a distributed system provides its functionality. This model is based on use-cases. As use-cases are expressed in a semi-formal format, they are converted into a formal format (Job) to make them understandable for autonomic managers. Taking the internal structure of a use-case into account, system malfunctioning can be associated with a use-case step (Task) that changes a state (State) or causes the occurrence of an event (Event). Section 5.1 explains these model elements as the common characteristics of the behaviour of distributed systems.

The structural model, presented in Section 5.2, describes the common aspects of the internal structure of software programs in a distributed system. Knowledge about the internal structure of a managed system is needed to identify which part of the software program causes a malfunctioning, and where in the software program remedy actions should be instrumented. The structural model provides abstractions to specify this knowledge.

5.1 Behavioural Model

Recall that the behavioural model presented in this thesis is based on use-cases. A use-case describes the response of a system to a given request. The system provides its response by a number of use-case steps executed by its structural elements. In the behavioural model, a Job represents a running use-case and a Task represents a use-case step.

To understand the response of a system to a request, an imaginary channel (job execution channel) is assumed in which tasks of a single job are executed by various structural elements (see Figure 5.1). Tasks within a single job are executed sequentially. All tasks in a chain of tasks manipulate data they receive from different data sources. A data source for a task can, for example, be a
database, a variable in a structural element, output of the previous task, or job input. A possible malfunctioning during execution of a job can be caused by: (1) a request (job input) that contains incorrect data, (2) incorrect data received from a data source such as a structural element or a database, (3) inappropriate manipulation of the data by a task, and/or (4) improper operation of computing resources on which a task runs (task’s environment).

Note that the output of one job can be the input to another job, creating a chain of job execution channels. The following sections explain the basic entities that play a role during execution of a job.

5.1.1 Job

The notion of Job is very close to the notion of use-case as described in Section 3.4. A Job is defined in a formal language with additional properties for management purposes. The ER diagram, depicted in Figure 5.2, shows the relations of Job.

Each Job has one or more Tasks, zero or more inputs, and at most one output. The tasks of a job manipulate input and result in output. A job’s inputs and output are specified as a State. The entity State is explained in detail in Section 5.1.3, for now it is sufficient to know that it represents a data item, and contains information about the data type of the item and the data source where it originates. Two additional Task types are added to a Job to check a job’s pre-conditions and post-conditions. To monitor the execution of the tasks of a job, sensors are associated to each Task.

A job can delegate sub-goals to other jobs, defining a parent-child relationship between jobs. A Job has zero or more children and zero or more parents (i.e., each job can invoke multiple jobs and be invoked by multiple jobs). This information is used by autonomic management to determine the context (the chain of invocations) of a system malfunctioning.

Sensors are associated with each Job to inform an autonomic manager about the start and end of a job, or about occurrences of events during execution of the job. Job’s effectors execute adaptation instructions changing a job’s behaviour (e.g., adding a new task or statement in the job’s code).

Four types of Jobs are distinguished corresponding to the four levels of use-cases distinguished in Section 3.5: SystemLevelJob, RunnableLevelJob, ComponentLevelJob, and ClassLevelJob, each with their own structural ele-
ments, symptoms, and hypotheses. For example, the tasks of a Runnable-LevelJob execute within ManagedRunnables (described in the next section), and RunnableLevelSYM and RunnableLevelHYP specify their associated symptoms and hypotheses respectively.

### 5.1.2 Task

A Job consists of a number of Tasks each representing a use-case step. As correct execution of a job largely depends on correct execution of its tasks, autonomic managers monitor their executions. As a result, autonomic managers can pinpoint a task (that is a more fine-grained behavioural element) as the root-cause of system malfunctioning.

The size of the actual code of tasks can vary. The code of one task may consist of thousands lines of programming statements, and the code of another task may consist of just one programming statement (such as an arithmetic operation). For instance, in most cases, the size of the actual code of a task belonging to a RunnableLevelJob is larger than that of a task belonging to a ClassLevelJob. From the viewpoint of an autonomic manager, there is no difference between tasks. The ER diagram, depicted in Figure 5.3, shows the common characteristics of a Task.

Each Task has zero or more inputs, and at most one output. The task’s input can be a job’s input, the value of a variable belonging to the structural element, the data item coming from a data source, or the output of one of the previous tasks. For example, the input of the runnable level task (8) (‘BusinessManager authenticates the user’), depicted in Figure 3.12, is the input of the Authentication
Realisation job, namely the user’s certificate.

Each Task is executed by exactly one structural element and monitored by one or more sensors. StateSensors monitor a task’s input and output states, and EventSensors monitor the occurrence of an event during execution of a task. Effectors adapt a task’s input or output state.

The following types of tasks are identified:

- A StateManipulationTask reads a data item from a data source, or writes a data item to a data source. Different tasks are distinguished for different sources. Examples are DataSourceStateManipulation, Managed-ElementStateManipulation, and JobInputStateManipulation. By monitoring these tasks, autonomic managers can relate system malfunctioning to an incorrect value of a specific state or states.

- A StateInteractionTask sends a state from one structural element to another structural element. This task is used when a managed use-case delegates an activity to another use-case. Autonomic managers can follow the flow of a state in the system by monitoring this type of task.

- An InvocationTask invokes a child job from a parent job. Monitoring this task makes it possible for autonomic managers to keep track of chains of use-case invocations. Switching control flow from one use-case level to another one is explicitly modelled (i.e., LowerInvocation, PeerInvocation, and HigherInvocation tasks). Invocation task’s property invocationType can be: local if both invoked and invoking jobs are in the same process space; synchRemote if the invoked and invoking jobs are in different process spaces and the invoking job blocks until the invoked job finishes its execution; or asynchRemote if the invoked and invoking jobs are in different process spaces and the invoking job does not wait for the execution of the invoked job to finish.
5.1 Behavioural Model

5.1.3 State

A State models a data item in a class, the status of a runnable, the parameters or local variables of a method, or the return value of a referenced job. States continuously change during execution of jobs. One of the most frequently encountered causes of system malfunctioning is related to an improper state change. Autonomic managers trace (1) where (in the system code) a state change occurs, and (2) when (during execution of which task) a state changes.

State has two attributes: name and type. When a State represents a variable occurring in the code of a managed system, the value of the name attribute is the qualified name of that variable. For example, the qualified name of an instance variable iv belonging to an object o is o.iv, and the qualified name of a static variable sv belonging to a class c is c.sv. The value of name is used to automatically find the corresponding variable in the code and instrument a sensor around it.

The type attribute makes it possible for autonomic managers to compare values, during runtime, to check whether a state contains the expected value. The following state types, based on the xml schema [21] data types, are distinguished: booleanType, integerType, doubleType, stringType, dateType, and timeType.

A job, during execution, can manipulate (or is affected by) different states in an execution channel. Autonomic managers are interested in the origin of these states to determine the source of an incorrect data item. Based on the data source a state originates, various states are identified:

- An InvocationResultState represents the value returned by an invoked job. If the value of this state is not as expected, the autonomic manager concludes that the problem may be caused by the referenced job and needs further information.

- A DataSourceState represents the value originating from a data source such as a database, file, message queue, message topic, or user interface including standard input/output console and web forms. Commonly, distributed systems obtain their data from a persistent data source or standard I/O console, store the data in a variable, and process the data. During system malfunctioning, this division helps identify possible data corruption within the specific data source.

- A JobInputState and JobOutputState represent the input and output value of a job during its execution, respectively. Note that the JobOutputState is different from InvocationResultState. The output value of a job can change as a result of certain problems (such as network problems) before this value can be received by the invoking job. Modeling these states makes it possible for an autonomic manager to check the pre- and post-conditions of a job.

- A ManagedElementState represents the values kept within structural elements. They describe the status of a process or thread (e.g., running, stopped) or a connector (e.g., broken connection, overloaded connection),
component variables (e.g. version, configuration parameters), static and dynamic variables declared in a class, and method parameters and local variables declared in a method.

5.1.4 Event

An Event is a notable occurrence at a particular point in time. There are two categories of events: managed events and management events. Managed events occur during execution of jobs and tasks, influencing the normal flow of execution. They can affect the behaviour of a distributed system, and cause system malfunctioning. Autonomic managers monitor events to obtain knowledge about a managed system. Examples of managed events are:

- those caused by execution of a specific task of a job, or
- those caused by a system’s environment that provides computing resources for job execution.

Management events are used to inform autonomic managers to take management actions. They are added to the code of the managed system by autonomic managers. Examples of management events are:

- events that inform autonomic managers about the life-cycle of a job, or
- events that warn autonomic managers to take a pre-defined management action.

The following Events are defined:

- An RunnableStartupEvent and RunnableShutdownEvent represent events that are executed to ascertain the startup and shutdown of a runnable, respectively. These events are used by autonomic managers to monitor the startup/shutdown order of various runnables. For example, suppose that an initialisation code in the BusinessIntegrator, described in Section 3.7, makes a connection to the DatabaseManager to fetch all user identification information from a database. The DatabaseManager is required to start before the BusinessIntegrator. Large distributed systems have many numbers of such dependencies among runnables that should be managed.

- A JobStartEvent and JobEndEvent represent events that are executed to ascertain when a job has been started and finished, respectively. Job execution takes place between the occurrences of these two events. JobStartEvent causes the autonomic manager responsible for that job to be activated.

- An InvocationEvent is executed to determine the moment at which a job references another job at a lower use-case level (LowerInvocationEvent), at the same use-case level (PeerInvocationEvent), or at a higher use-case level (HigherInvocationEvent).
5.2 Structural Model

- An **ExceptionEvent** is executed to determine an abnormal and unexpected happening during execution of a job. The **NullPointerEvent** (referencing an uninitialised object) and **NoSuchFieldEvent** (referencing a non-existing object field) exceptions are examples of the exception event.

- A **TimerExpirationEvent** triggers autonomic managers to start their timer-based activities. For example, an autonomic manager can use this event to check the availability of a runnable at regular intervals.

### 5.2 Structural Model

Distributed systems are most often composed of one or more sub-systems, each of which in turn is composed of a number of components. Components either contain other (sub)components or a number of classes. Classes may contain other (sub)classes and a number of methods. Each of these elements (**structural elements**) can be considered to be a logical unit containing system code.

The common properties of the different structural elements are modelled as **ManagedStructuralElement**. One of the common properties of structural elements is that they contain code with which use-case steps are executed. The other common property is that they can all contain one or more states that keep data needed for the execution of use-case steps. The **ManagedStructuralElement** entity in the structural model contains properties which reflect the mentioned properties of the structural elements.

The following managed structural elements are distinguished: **ManagedSystem**, **ManagedRunnable**, **ManagedComponent**, and **ManagedClass**. As shown in Figure 5.4, these elements correspond to the use-case levels introduced in Chapter 3.

![Figure 5.4: Various types of managed structural elements that correspond to the use-case levels.](image_url)
88 The System Model

Each of these structural elements contains one or more other structural elements, and has the common property entryPoint for activation. There are also two other managed structural elements: ManagedMethod and ManagedConnector. These are atomic, meaning that they do not contain other managed structural elements. The following sections explain all different types of managed structural elements and their relationships. The UML structural diagrams illustrate the relationships between different structural elements.

5.2.1 Managed System

Usually a number of related use-cases, addressing the specific needs of an organisation’s processes and data flow, are grouped and packaged as one system. Many enterprises define a clear boundary for each system, and organise their software maintenance and fault handling processes around these systems. There are dedicated teams for maintaining specific software systems. A ManagedSystem in the proposed structural model represents a system, a logical entity containing code with which a collection of related behaviours (use-cases) is executed.

As explained in Section 3.5, all use-case steps in system level use-cases are executed by one structural element, a ManagedSystem. Autonomic managers can pinpoint this structural element as the source of system malfunctioning during execution of system level use-cases.

![Figure 5.5: A ManagedSystem contains one or more ManagedRunnables, and one ManagedMethod.](image)

A ManagedSystem is composed of a number of ManagedRunnables. Each ManagedSystem contains one special ManagedMethod which serves as its entry point (see Figure 5.5). Managed method starts the executions of all runnables belonging to that system. Autonomic managers use this entry point to monitor the start order of the execution of runnables (i.e., to make sure that runnable $r_1$ starts after/before runnable $r_2$). For example, if an Application Server requires initialisation data, stored in a database, and it starts before the Database Server then the Application Server will fail to start.

5.2.2 Managed Runnable

A computer program, written in a programming language, is compiled into a set of instructions. The operating system of a computer loads these instructions into computer memory and starts a process to execute the computer program. Within a process, a computer program can split itself into two or more concurrently running pseudo-processes which share the same resources allocated to their process. These
5.2 Structural Model

pseudo-processes are called *execution threads*. Processes and threads have certain properties (such as host, port, process-id, thread-id, status) that are of importance to management. A ManagedRunnable models a process or an execution thread.

Runnable level use-cases describing the internal behaviour of a system as interactions between runnables are executed by ManagedRunnables. Autonomic managers at this level can pinpoint these structural elements as the source of system malfunctioning during execution of runnable level use-cases.

![Figure 5.6: Relationship between the composite and atomic ManagedRunnables.](image)

A ManagedRunnable is either atomic or composite (see Figure 5.6). A CompositeManagedRunnable contains a combination of AtomicManagedRunnables and/or CompositeManagedRunnables. As a result, the related processes (or threads) can be bundled and represented in the model as one (composite) managed runnable. Using CompositeManagedRunnable, it is possible to model distributed middleware software systems such as a Java 2 Enterprise Edition (J2EE) application server (e.g., IBM WebSphere Application Server [92] or JBoss Application Server [100]). It is also possible to model a structurally complex composite web service [151] that aggregates a number of web services (atomic or composite) - running on different remote machines - according to a certain composition pattern.

![Figure 5.7: An AtomicManagedRunnable contains one ManagedMethod, one or more ManagedComponents, and zero or more ManagedConnectors which bind the atomic managed runnables to each other.](image)

Figure 5.7 shows that an AtomicManagedRunnable is composed of one or more ManagedComponents, and contains a special ManagedMethod that represents the main method. In most programming languages, this method serves as the entry point of a program, and it is where a program starts its execution. Autonomic managers use this entry point to monitor the flow of execution from a runnable to its components.
5.2.3 Managed Connector

Atomic managed runnables can be connected with each other by means of a ManagedConnector using specific protocols. Each specific protocol defines its own rules governing the syntax, semantics, and transfer of data between two sub-systems. An autonomic manager possessing the knowledge about a specific protocol, used by two runnables, is able to properly manage the connection between these runnables.

As explained in Section 3.6, a complex use-case references simple use-cases to delegate certain sub-goals to referenced use-cases. The use-case step type (i.e., InvocationTask), that describes this reference, is executed by a piece of software code that implements the connection between two runnables. The ManagedConnector models this piece of software code. This structural element can be pinpointed by autonomic managers, similar to system administrators, as the source of connection problems.

The following entities, containing meta-information about the corresponding communication protocols, are part of the structural model:

- A DataAccessProtocol represents the protocol used between a runnable executing business logic and a Database Management System (DBMS) to persist data permanently on a storage device. This entity is used to model the most widely used protocols Java Database Connectivity (JDBC) [176] and Open Database Connectivity (ODBC) [134].

- A FileOrientedProtocol represents the protocol usually used by legacy sub-systems to communicate with each other by reading or writing a shared file in a proprietary format.

- A MessageOrientedProtocol makes it possible for runnables to asynchronously exchange messages in a standard format using either the message-queue mechanism or the message-topic mechanism. The message-queue mechanism makes it possible for two runnables to exchange messages directly (point-to-point), and the message-topic mechanism provides a way to publish messages to (or consume from) multiple runnables at the same time.

- A StreamOrientedProtocol transfers data on demand, in real-time, as a continuous stream of bytes.

- A WebOrientedProtocol is used for communication through Internet. The most frequently used protocol Hypertext Transfer Protocol (HTTP) is modelled by this entity.

- An RPCProtocol (Remote Procedure Call) and ROIProtocol (Remote Object Invocation) are used to invoke functionality in external address spaces. The most commonly used protocols Open Network Computing Remote Procedure Call (ONC RPC) [174] and Java Remote Method Invocation (JRMI) [175] are modelled by these entities.
5.2.4 Managed Component

A ManagedComponent models a software component (or a library). A software component groups a set of related classes (code) to provide specific business functionality (e.g., retrieving customer profile) or specific technical utility (e.g., logging). Each software component has a set of service access points (interface methods) that hides the implementation details of the component from its clients.

Component level use-cases describing the internal behaviour of a system as communication between components are executed by ManagedComponents. Autonomic managers at this level can pinpoint these structural elements as the source of system malfunctioning during execution of component level use-cases.

![Figure 5.8: Relationship between the composite and atomic ManagedComponents.](image)

A ManagedComponent is either atomic or composite (see Figure 5.8). A CompositeManagedComponent contains a combination of AtomicManagedComponents and/or CompositeManagedComponents. Composite managed components model more complex structural elements consisting of a set of related libraries and reusable software components. An Object-Relational Mapping (ORM) framework (such as Hibernate [99]), containing both the core elements of the framework and other utility libraries such as logging, is modelled as a CompositeManagedComponent.

![Figure 5.9: A ManagedComponent contains one or more ManagedMethods, and one or more ManagedClasses.](image)

Figure 5.9 shows the relationship between an atomic managed component with its managed classes. An AtomicManagedComponent contains one or more dedicated ManagedMethods, and one or more ManagedClasses. The managed methods, that represent the interface methods of a component, form the entry-points of an atomic managed component. Autonomic managers use these entry points to manage the interaction of a component with other components or with its classes.
5.2.5 Managed Class

A ManagedClass models a module or a class. A module in imperative or scripting programming paradigm groups a number of related variables and functions (procedures or subroutines). A class in object-oriented programming paradigm abstracts the real-world objects (e.g., employee, currency, customer) by grouping related fields and methods. Fields contain data that are manipulated by methods. Each method contains a collection of statements (instructions) that can access the data stored in a field and change the state of a class.

Class level use-cases describing the internal behaviour of a system as invocations between methods are executed by ManagedClasses. Autonomic managers at this level can pinpoint these structural elements as the source of system malfunctioning during execution of class level use-cases (i.e., they can determine which method of which class incorrectly manipulates a certain system state).

![Diagram of Managed Classes](image)

**Figure 5.10:** Relationship between the composite and atomic ManagedClasses.

A ManagedClass is either atomic or composite (see Figure 5.10). Similar to the recursive definitions of managed runnables and managed components, a CompositeManagedClass contains a combination of AtomicManagedClasses and/or CompositeManagedClasses. The CompositeManagedClass can be used to model the hierarchical relationship between a class and its sub-classes.

An AtomicManagedClass contains a number of ManagedMethods, and one dedicated managed method which represents the class’s constructor method and forms the entry-point of an atomic managed class. A constructor method is called automatically when an instance of a class is created. Autonomic managers use this entry point to monitor the state of a class during its creation.

5.2.6 Managed Method

A method in object-oriented programming languages (such as Java) is associated with a class, and usually consists of a sequence of programming statements to access and manipulate the state of the class. A method accepts a set of input parameters, and returns a possible output. A ManagedMethod models a method. A ManagedMethod can also represent a function, procedure, or a subroutine in imperative programming languages (such as C or Pascal) or scripting languages (such as Perl or Bash). The parameters and local variables are assumed to be the
states of a ManagedMethod. Autonomic managers monitor these states to manage the execution of the statements contained in a method.

5.3 Summary

The model of a managed system presented in this chapter contains both behavioural and structural models. The behavioural model describes the way a distributed system provides its functionalities, and the structural model describes the internal structure of software programs for a distributed system. The behavioural model contains jobs, tasks, states, and events, and the structural model contains managed structural elements.

A job represents a use-case to be executed within the managed structural elements, and consists of a number of tasks each representing a use-case step to manipulate states. Manipulation can cause system malfunctioning. Autonomic managers need to know which part of the software program causes the malfunctioning, and where in the software program the remedy actions should be instrumented. By combining the behavioural and structural model, autonomic managers can pinpoint the location of possible root-causes of system malfunctioning.
Chapter 6

Self-Management Knowledge Representation

The term *self-management knowledge* refers to (1) knowledge a system has of its internal structure and its dynamic behaviour, and (2) meta-knowledge a system has to evaluate and adjust its own behaviour. This knowledge needs to be represented in a formal way that supports interpretation by an autonomic manager. In this thesis the knowledge is expressed in a formal language that is processable and understandable by software modules.

Representation of self-management knowledge is important for two reasons. First, domain experts use representations to provide knowledge regarding a specific system to be managed. Second, representations can be used to generate code for a specific autonomic manager. A considerable number of formal languages have been used for knowledge representation (see [44] for an overview of traditional and web-based languages, their expressiveness and reasoning capabilities). This chapter discusses the choice of a suitable language for self-management knowledge representation.

6.1 Knowledge Representation Requirements

This section discusses a number of important requirements [82] for representation of self-management knowledge in distributed environments.

6.1.1 Knowledge Locality & Modularity

A distributed system is a system with multiple computing elements distributed across a network. Efficient self-management mandates co-location of self-management knowledge, that relates to a specific computing element, and the computing element itself. Co-location implies that the system as a whole needs to be able to reason with such distributed knowledge.
Assume that a managed system is composed of a number of Web Services distributed on the web, each with its own self-management knowledge, and each provided by a different vendor. Such knowledge is needed both at the level of each of the Web Services and at the level at which they are combined.

### 6.1.2 Knowledge Reasoning

Self-management knowledge does not only contain concepts, but also rules to reason about concepts, and meta-rules to reason about these rules. Rules are used in a running self-management system to analyse a system’s current status, detect abnormal behaviour, and repair system malfunctioning. Self-management concepts and rules are distributed across network. There is a need for a rule language that supports reasoning with distributed knowledge.

### 6.1.3 Knowledge Acquisition

To support the creation and maintenance of the self-management knowledge, two non-functional requirements hold:

- **Tool availability** - The availability of tools to support knowledge acquisition in distributed environments is essential.
- **User acceptance** - The language used for self-management knowledge specification must be intuitive and comply with standards.

### 6.2 Choice for Knowledge Representation

As the self-management model is driven by use-cases and UML/OCL [157] is used to describe use-cases, UML/OCL is one of the options considered. However, UML/OCL has two shortcomings: (1) it is not possible for a concept specified in UML/OCL to refer to another concept that is located (distributed) elsewhere on the network, (2) UML/OCL does not have logical reasoning capabilities (i.e., it is not possible to express logical statements in UML/OCL).

An alternative for self-management knowledge representation is a language based on *first-order logic* [166]. In principle, first-order languages can be used to specify every finite system. However, (1) it is very cumbersome to specify complex concepts and inheritance relations over the concepts (subsumption) in first-order languages, (2) it is difficult to add the notion of *sort* (type) to the formalism of first-order logic in order to categorise a domain into groups, and (3) there is to our knowledge no first-order language available that provides a mechanism to refer to distributed concepts located on the network.

Another option is the family of knowledge representation languages known as *description logics* [139] which have evolved from a combination of *semantic net-
6.3 Semantic Web Overview

works\(^1\) [167] and *high-order logic*\(^2\) [30]. Ontologies, implemented in description logic languages, are increasingly used to this end. For example, Stojanovic et al. [173] use ontologies to describe resources and changes in the state of a resource in a correlation engine, and Jannach et al. [96] use ontologies to describe multimedia resources and transformation actions in a multimedia adaptation engine. According to the principles of knowledge representation presented in [49], self-management knowledge representation can be viewed as a set of ontological commitments, i.e., agreements about the concepts and their relations, as discussed in Chapters 4 and 5.

This thesis has chosen to represent self-management knowledge as ontological commitments in the description logic language OWL and the rule language SWRL. The following section provides an overview of the Semantic Web languages OWL and SWRL, and the argumentation for the choice.

6.3 Semantic Web Overview

The central idea of the *Semantic Web* initiative [18] is to augment the current web with formalised knowledge to make information on the web machine-processable. The Semantic Web relies on:

- **ontologies** by which the domain concepts, concept hierarchies, and concept relationships can be expressed, and
- **logical reasoning** by which new conclusions can be drawn after combining data with ontologies.

Semantic Web ontologies are expressed in a description logic language such as *Ontology Web Language (OWL)* [16]. Logical reasoning rules are expressed in *Semantic Web Rule Language (SWRL)* [90] (a W3C proposal).

In the Semantic Web hierarchy of languages, shown in Figure 6.1, *Uniform Resource Identifier (URI)* is depicted as the lowest layer. A URI is a string of characters used to identify an abstract or physical resource [17]. A resource can be anything that is located anywhere and has an identity (e.g., an electronic document, an image, a service). The main purpose of identifying resources with a URI is to find, exchange and combine data about the identified resource across the Web.

The layer above the URI layer is the *Extensible Markup Language (XML)* [23]. XML is an extensible specification language for creating custom markup languages. A well-formed XML document conforms to the XML syntax rules. Additionally, a valid XML document conforms to semantic rules defined in an *XML schema* [21] document.

---

\(^{1}\)A semantic network represents semantic relations between the concepts as a graph. The vertices of the graph represent concepts, and the edges represent their relations.

\(^{2}\)Languages based on higher-order logic provide more possibilities to express more complex concepts through allowing to define quantifications over predicates, and introducing sets of individuals.
The layer above the XML layer is the Resource Description Framework (RDF) [120]. RDF provides a common data model for describing resources, identified by URIs, on the Web. A graph of resource descriptions contains a collection of RDF statements about web resources as subject-predicate-object expressions. The subject denotes the resource, and the predicate denotes characteristics of the resource and expresses a relationship between the subject and another resource (the object). RDF is designed to be interpreted by computers. RDF statements are written in XML. RDF schema (RDFS) [27] is a language for describing ontologies in RDF, and it provides mechanisms to describe groups of related resources (resource classes) and the relationships between these resources (resource properties). In analogy to XML schema language, in which semantic rule definitions are written in XML, RDFS ontology descriptions are written in RDF.

OWL is the layer above RDF and RDFS. OWL is based on the DARPA Agent Markup Language (DAML) [47] effort. OWL combines the expressive power of description logics with the simplicity and distributive nature of RDF. SWRL extends the set of OWL axioms with Horn-like rules to enrich OWL ontologies. A rule is an implication between an antecedent (body) and consequent (head). Atoms in these rules consist of OWL concepts, properties, individuals, data values, or variables. The following sections describe some of OWL’s features.

### 6.3.1 OWL Concepts & Properties

A concept (Class) in OWL is interpreted as a set of individuals, and a property relates individuals of different concepts to each other. In fact, each property is a mapping between a member of a domain and a member of a range. The domain of a property consists of the individuals of one or more concepts. This is also true for the range of a property.

There are two types of properties: ObjectProperty and DatatypeProperty. The members of the range of the first type are the members of an OWL class, and the members of the range of the second type are data literals (members of a primitive type such as integer, dateTime, etc.).

Examples of an OWL class are Person and University. Examples of an OWL individual, belonging to these OWL classes, are Frank and TUDelft. An example OWL Object property is isEmployeeOf. The domain and range of this property respectively are the Person and University classes. An example OWL Datatype property is hasAge of which the domain is the OWL class Person and the range is the data type integer.
6.3 Semantic Web Overview

6.3.2 OWL Cardinality Restrictions
In OWL, *cardinality restrictions* are used to constrain the number of values of a particular relationship. Cardinality restrictions are defined on Object and Datatype properties. They specify either the *minimum* number, the *maximum* number, or the *exact* number of a specific relationship of an individual. An example of a maximum cardinality restriction is to restrict the property `hasParent` such that the individuals of the OWL class `Person` have at most two parents.

6.3.3 OWL Value & Existential Restrictions
These restrictions limit which values (instead of how many values) of the range of the property can be used by instances of a concept. The *value restriction* states that all members of the range of a particular property should be of a certain type, and the *existential restriction* states that at least one member of the range of the property should be of a certain type.

Consider two OWL classes `Student` and `Game`. A value restriction on the property `playsGame` can be used to specify that if a student plays a game then he/she plays one of the games from the OWL class `Game`. Consider two OWL classes `Student` and `Teacher`. An existential restriction on the property `hasParent` can be used to specify that the parent of at least one student is a teacher.

6.3.4 OWL Consistency Check
The consistency is checked by an OWL reasoner such as Pellet [164] or Racer [77]. The reasoner checks to see whether the instances of all mandatory concepts and the relationships between them have been correctly specified. For this purpose, an OWL reasoner utilises the meta-level information available in the *Assertional Box* (ABox) and *Terminological Box* (TBox). The ABox contains instances of concepts and instances of the relationships between these concepts. The TBox contains definitions of concepts, relationships between them, and different restrictions on those relationships. A model in OWL is said to be *consistent* if all sentences in the ABox are consistent with respect to the TBox of the model.

6.3.5 Requirements Satisfaction
This section argues that the Semantic Web languages OWL and SWRL satisfy the requirements for self-management knowledge representation specified before in Section 6.1.

**Knowledge Locality & Modularity**
The requirement of locality and modularity is satisfied because the Semantic Web languages support the use of URIs. URIs make it possible to identify and refer to resources stored at different locations. Local knowledge consists of self-management concepts and rules which describe the internal structure and the
behaviour of each computing element of a distributed system. These concepts and rules are the resources to which URIs refer.

Knowledge Reasoning

The knowledge reasoning requirement is satisfied because SWRL is specifically designed to reason about OWL concepts. SWRL is closely integrated with OWL: rules in SWRL use OWL concepts. A rule in SWRL can refer to an OWL concept or to an OWL property located on the network, through a URI. Also, a rule in SWRL can refer to another SWRL rule located on the network, through a URI. For example, SWRL built-in rules such as lessThan(x, y), located at the internet address http://www.w3.org/2003/11/swrlb, can be used within the SWRL rules located at a different internet address.

Knowledge Acquisition

Both non-functional requirements defined in Section 6.1.3 are satisfied by Semantic Web languages with respect to knowledge acquisition:

- The tool availability requirement is satisfied: both the open-source community and commercial companies provide various Integrated Development Environments (IDEs), plugins to other existing IDEs, Java APIs, and tools and techniques for checking the syntax and consistency of OWL documents.

- The user acceptance requirement is satisfied: the Semantic Web languages comply with W3C standards and have common and relevant features with Unified Modeling Language (UML). UML is the de facto industrial standard for software development. The Ontology Definition Metamodel (ODM) [81] defines the relationship between the relevant features of UML and OWL.

In conclusion, the choice for Semantic Web languages to represent self-management knowledge is justified with the requirements formulated in Section 6.1. The next section explores the use of these languages to express the knowledge needed in a self-management framework.

6.4 Self-Management Ontology

In this thesis, the Semantic Web languages OWL and SWRL are used to express both generic and domain models. The generic model is the formal representation of the self-management concepts for all distributed systems, as described in Chapters 4 and 5. The domain model is the model of a specific managed system. This section illustrates the ontology for parts of the generic model in OWL. The ontology of the generic model is modular and extensible. It is composed of a number of sub-ontologies each of which contains their own reusable knowledge.

6.4 Self-Management Ontology

6.4.1 Autonomic-Manager Sub-Ontology

The autonomic-manager sub-ontology has been defined to describe autonomic managers (and their process results) discussed in Section 4.7. The emphasis in this section is on the definition of cardinality restrictions in OWL.

```
<owl:Class rdf:ID="AutonomicManager">  
  <rdfs:subClassOf>  
    <owl:Restriction>  
      <owl:onProperty><owl:ObjectProperty rdf:ID="job"/></owl:onProperty>  
      <owl:cardinality rdf:datatype="&xsd;int">1</owl:cardinality>  
    </owl:Restriction>  
  </rdfs:subClassOf>  
  <rdfs:subClassOf>  
    <owl:Restriction>  
      <owl:onProperty><owl:ObjectProperty rdf:ID="subAutonomicManagers"/></owl:onProperty>  
      <owl:minCardinality rdf:datatype="&xsd;int">0</owl:minCardinality>  
    </owl:Restriction>  
  </rdfs:subClassOf>  
  ...  
</owl:Class>  
```

Figure 6.2: Part of the autonomic-manager sub-ontology showing the specification of cardinality restrictions in OWL.

Figure 6.2 shows the definition of the AutonomicManager concept and two of its properties. Both properties are ObjectProperties. The restriction on the first property limits the number of jobs of an autonomic manager to exactly one. The restriction on the second property specifies that an autonomic manager may have zero or more children. The OWL statement at the end of the figure depicts that the range of the first property, Job, is specified in another sub-ontology (behav-model) located on the web.

6.4.2 Behavioural-Model Sub-Ontology

The behavioural-model sub-ontology has been defined to describe the dynamic behaviour (use-case realisation) of a managed system. This sub-ontology consists of the specification of a collection of jobs, tasks, states, and events discussed in Section 5.1. This section focuses on the definition of concept inheritance in OWL.

Figure 6.3 shows the definition of the Task concept and one of its sub-concepts. The restriction on the property managedElement indicates that any Task is to be executed inside exactly one managed structural element. The Invocation task, which represents the invocation of a child job from a parent job, is a sub-concept of Task. This means that it inherits all properties of a Task. Moreover, it introduces a new property invokedJob that represents the child job. The specification also indicates that a property can be shared by different concepts. For instance, the managedElement property is also used by ElementStartupEvent to indicate the runnable that is started.
6.4.3 Structural-Model Sub-Ontology

The structural-model sub-ontology has been defined to describe the internal structure of a managed system. This sub-ontology consists of a specification of all different types of structural elements discussed in Section 5.2. The focus is on the value restrictions and recursive concept definitions in OWL.

The OWL definitions in Figure 6.4 are based on the diagram depicted in Figure 5.8. The property subElement is a shared property used by both AtomicManagedComponent and CompositeManagedComponent. As an atomic managed component contains one or more managed classes, the OWL value restriction is used to restrict the range of the subElement to ManagedClass. The range of the subElement for the composite managed component is restricted to ManagedComponent. The CompositeManagedComponent definition also shows an example of the recursive definitions in OWL. On the one hand, the CompositeManagedComponent is a sub-concept of the ManagedComponent, and on the other hand, the range of the property subElement is restricted to the parent concept.

6.4.4 Analyser Sub-Ontology

The analyser sub-ontology has been defined to describe how an analyser analyses job execution, and how incorrect behaviour of a job can be expressed as symptoms.
6.4 Self-Management Ontology

The focus in this section is on DatatypeProperty in OWL.

Figure 6.4: Part of the structural-model sub-ontology showing the specification of value restrictions and recursive definitions in OWL.

Figure 6.5: Part of the analyser sub-ontology showing the specification of DatatypeProperty in OWL.

Figure 6.5 illustrates the definition of Symptom and one of its properties existInspectivePlan. This property indicates whether there is an inspective plan with which the occurrence of a symptom can be clarified. The property is an OWL DatatypeProperty and its range is the XMLSchema's boolean type.

6.4.5 Diagnoser Sub-Ontology

The diagnoser sub-ontology has been defined to describe a diagnoser, its SWRL rules, and hypotheses. The emphasis in this section is on the definition of necessary and sufficient [89] conditions in OWL. These conditions limit the membership of an OWL class to having dedicated individuals.
Figure 6.6 shows the definition of the Ternary class. The members of this class are limited to the OWL instances (individuals in OWL) Pos, Neg, and Unknown. These OWL instances are used to indicate that the result of the validation of a hypothesis is positive, negative, or unknown. The Ternary class is also used to denote the value of a symptom, signifying its occurrence.

The diagnoser sub-ontology also includes generic hypothesis selection, validation, evaluation, and determination rules. These rules are expressed in SWRL. An example of a rule is that a hypothesis is not selected for validation purposes if it has already been evaluated. Figure 6.7 depicts this rule in SWRL. The highest level SWRL class is swrl:Imp that represents a single SWRL rule. This class contains a consequent part (swrl:head) and an antecedent part (swrl:body), each containing a list of rule atoms.

6.4.6 Sensor Sub-Ontology

The sensor sub-ontology has been defined to describe the different sensor types with which runtime information from a running managed system is retrieved. In this section, the focus is on the definition of an OWL property for which the range is not known in advance. Because sensors in the code of a managed system monitor values of states of several different types (boolean, string, integer, etc.), it is not possible to determine the range of the observedValue property of the Sensor in advance.

---

4SWRL has two different syntaxes: the XML Concrete syntax and the Human Readable syntax [90]. The example rule depicted in Figure 6.7 is in XML Concrete syntax. The XML Concrete syntax is a combination of the OWL syntax with the RuleML XML syntax [90]. SWRL statements in this syntax can be combined with OWL concepts and properties. The example rule in Human Readable syntax is as follows:

Hypothesis(?hy) ∧ tried(?hy, true) → toBeFocussed(?hy, false)
6.4 Self-Management Ontology

<swrl:imp rdf:ID="hypSelRule-4">
  <swrl:head>
    <swrl:atomList>
      <swrl:datavaluedPropertyAtom>
        <swrl:propertyPredicate rdf:resource="#toBeFocused"/>
        <swrl:argument1 rdf:resource="#hy"/>
        <swrl:argument2 rdf:datatype="&xsd;boolean">false</swrl:argument2>
      </swrl:datavaluedPropertyAtom>
    </swrl:atomList>
  </swrl:head>
  <swrl:body>
    <swrl:atomList>
      <swrl:classAtom>
        <swrl:argument1 rdf:resource="#hy"/>
        <swrl:classPredicate rdf:resource="#Hypothesis"/>
      </swrl:classAtom>
      <swrl:datavaluedPropertyAtom>
        <swrl:propertyPredicate rdf:resource="#tried"/>
        <swrl:argument1 rdf:resource="#hy"/>
        <swrl:argument2 rdf:datatype="&xsd;boolean">true</swrl:argument2>
      </swrl:datavaluedPropertyAtom>
    </swrl:atomList>
  </swrl:body>
</swrl:imp>

Figure 6.7: Part of the diagnoser sub-ontology showing one of the generic rules in SWRL’s XML Concrete Syntax.

<owl:class rdf:ID="Sensor">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty><owl:DatatypeProperty rdf:ID="observedValue"/></owl:onProperty>
      <owl:maxCardinality rdf:datatype="&xsd;int">1</owl:maxCardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:class>

<owl:DatatypeProperty rdf:ID="observedValue"/>

Figure 6.8: Part of the sensor sub-ontology showing the specification of observedValue in OWL.

OWL assumes that the range of a DatatypeProperty such as observedValue can be any XML Schema type [21] if the range is not pre-defined. Figure 6.8 shows the definition of the observedValue property. The domain of this property is the Sensor class, and the range of the property is empty.
6.5 Related Work

Recall that self-management knowledge should be represented in a formal language in order to be understood by autonomic management. Unfortunately, there is no standard language for expressing self-management knowledge. Different researchers have created their own proprietary languages. Fortunately, most of these languages are based on XML and RDF. The XML and RDF languages are open standards, web enabled, extensible, and platform independent. They enable distributed systems, running on different platforms across the web, to share information easily. The following briefly describes how different autonomic approaches utilise these languages.

**XML Based Languages**

XML is a general purpose language upon which custom languages can be created. XML-Based Architecture Description Language (xADL) [48, 111] and Architecture Description Markup Language (ADML) [169] are XML-based mark-up languages for describing software and system architectures. A number of basic tags which are introduced in these languages are: `<Architecture>`, `<Component>`, `<Connector>`, `<Topology>`, and `<Expression>`. Domain experts specify system structure in one of these languages making an abstract architectural model of a managed system. This architectural model is used by an architecture-based autonomic management to evaluate the managed system for constraint violation during runtime.

Examples of XML-based policy languages used by policy-centric autonomic computing approaches are Autonomic Computing Policy Language (ACPL) [1], Autonomic Computing Expression Language (ACEL) [1], and AGILE [7]. The first and second languages are used within the IBM’s Policy Management for Autonomic Computing (PMAC) platform. Each policy, consisting of conditions, actions, priority, and role, is expressed in ACPL. Policy conditions are expressed in ACEL. ACEL defines nine primitive types and various types of operators such as arithmetic functions, string functions, and calendar operations. The autonomic manager in PMAC is a policy-based manager that obtains its policies from the policy storage to provide policy guidance to managed resources. The third language (AGILE) facilitates dynamic adaptations of the policy configuration of autonomic systems. A policy, written in AGILE, is able to change its own behaviour through the use of indirect addressing. The monitoring library of AGILE automatically attaches monitoring logic to each numeric variable and silently monitors successive sample values. So, various temporal characteristics of an input stream (such as mean value, largest value, etc.) are collected and represented as properties that can be directly incorporated into policy logic.

**RDF Based Languages**

Recall that RDF itself is based on XML and provides a common data model for describing resources on the web. Stojanovic et al. [173] propose using RDF-based
ontologies for representing monitored data that are analysed by correlation engines implementing the MAPE model. According to them, ontologies provide a shared understanding of the managed domain, and therefore they facilitate interoperability between different correlation engines that together are responsible for managing systems in heterogeneous distributed environments.

Jannach et al. [96] use ontologies to describe multimedia resources and transformation actions in a multimedia adaptation engine. Two important components of their adaptation engine are: adaptation plan and adaptation service. The first component contains a suitable sequence of transformation steps, and the second one contains adaptation methods that can be applied on the multimedia content. They use the XML-based MPEG standards as domain ontology to represent various multimedia resources, and the RDF-based Semantic Markup for Web Services (OWL-S) [130] together with SWRL to represent adaptation services.

Keeney et al. [106] use the RDF-based ontologies to represent autonomic elements in an autonomic network in order to enable ontological reasoning. They derive OWL classes and properties from existing management information represented as Common Information Model (CIM) [55] objects. These OWL classes are used for two purposes: for defining runtime policies constraining the behaviour of an autonomic element, and for providing management capabilities that are specified as OWL-S services.

6.6 Summary

The representation of self-management knowledge is important because domain experts use the representation to provide knowledge regarding a specific system to be managed. So, on the one hand, the representation must be powerful enough to express and reason about distributed concepts regarding a distributed system, and on the other hand, there must be user friendly tools for the representation to support knowledge acquisition. Additionally, it must be possible to use the representation to generate code for a specific autonomic manager.

This chapter presents a number of important requirements for representing self-management knowledge in distributed environments, and argues that the Semantic Web languages OWL and SWRL together satisfy the requirements. Parts of the sub-ontologies of the ontology of the self-management model in OWL have been presented for the purpose of illustration.
Chapter 7

Illustrative Scenarios

This chapter illustrates how the self-management framework actually works for the Trading System\(^1\) introduced in Chapter 3. The Trading System is an existing system used by a banking enterprise. The focus is on how the self-management framework proposed in this thesis could be used in practice. Section 7.1 explains the methodology used to apply the framework for self-diagnosis.

Two types of failure are examined in two separate case studies. The purpose of the first case is to illustrate how the framework and its generic diagnostic rules can be used to analyse and diagnose the root-cause of multiple failures automatically (Section 7.2). In this case study, multiple failures occur during simultaneous execution of two single-level (runnable level) use-cases. The purpose of the second case is to illustrate how multi-level diagnosis is achieved by interaction between autonomic managers (Section 7.3). In this case study, a failure occurs during execution of a multi-level use-case. A complex use-case is divided into a hierarchy of simpler use-cases, and the corresponding autonomic managers are related with each other accordingly. Both types of failure are illustrated for use-cases and fault scenarios\(^2\) encountered in practice.

7.1 Methodology

Deployment of the proposed self-management framework for an existing distributed system entails\(^3\) specification of a self-management ontology of the system

\(^1\)Note that it is not the aim of this chapter to discuss whether or not the system or its sub-systems have been designed and implemented appropriately.

\(^2\)These fault scenarios have been encountered in the author’s extensive practical experience as a system architect in a banking enterprise.

\(^3\)Deployment of the framework also entails: (a) generation of autonomic management for an existing system using the code generation tool of the framework, (b) instrumentation of the specified sensors and effectors in the code of the system using the code instrumentation tool of the framework, and (c) preparation of the execution environment by linking the generic software libraries, provided by the framework, with the generated code, and bootstrapping different software processes in the appropriate order. The code generation and instrumentation are explained in Chapter 8.
using the generic self-management ontology. A specific self-management ontology requires specification of:

1. **Managed System Model Specification** - The unit of system model specification is a use-case. The behavioural and structural models of each use-case are specified in OWL format. The use-case model consists of the specification of a Job, its Tasks and ManagedStructuralElements, and Sensors that monitor the Job and its Tasks.

2. **Autonomic Manager Specification** - An AutonomicManager including its Analyser, Diagnoser, Planner, and PlanTranslator are specified for each use-case. Moreover, the parent-child relationships between autonomic managers are specified.

3. **Symptoms & Hypotheses Specification** - Each autonomic manager has its own set of Symptoms and Hypotheses for the Job it manages. The Symptoms, Hypotheses, and their many-to-many relationships are specified.

4. **SWRL Rules Specification** - The specific SWRL rules used by the AutonomicManager to determine a diagnosis are specified. Generic SWRL rules are provided by the framework.

Specification of model elements (such as a symptom or an autonomic manager) corresponds to the self-management elements introduced in Chapters 4 and 5. Note that all specification code in OWL, shown in the coming sections, is generated by means of the Protege-OWL editor [172]. The Protege-OWL editor provides a graphical user interface with which to specify OWL classes, properties and their instances (individuals). OWL classes and properties for self-management of all distributed systems (the generic part of the self-management model) are pre-defined. They are provided by the framework. For the self-management of a specific distributed system, domain experts specify domain-specific instances of the self-management classes and properties. For the sake of brevity, in the coming sections, only parts of the specifications in OWL are shown.

During runtime, autonomic management logs data regarding its autonomic process (receiving sensor values, inferring symptoms, executing inspective plans, determining a diagnosis, and communicating the diagnostic result from a child to the parent autonomic manager) to a log file. This log file makes it possible to study a specific execution process related to autonomic management. In the coming sections, parts of this log file are shown to display the flow of activities of the autonomic managers.

In the following sections, the above methodology is depicted for two case studies. The first case study focuses on self-diagnosis of multiple failures during execution.

---

Footnotes:

4. Note that the framework does not require specification of models of all use-cases of a system.

5. The complete code of both experiments including their autonomic management can be downloaded from [http://www.iids.org/research/self-management/self-management-experiments.zip](http://www.iids.org/research/self-management/self-management-experiments.zip).
of two single-level use-cases. The second case study focuses on self-diagnosis of a failure during execution of a multi-level use-case.

7.2 Case 1: Single-Level Use-Case Management

This section describes how the proposed self-management approach is applied to management of two single level use-cases of the Trading System: Authentication Realisation and Payment Status. This section describes the two use-cases, and explains how a generic (domain independent) diagnostic rule pinpoints the root-cause of multiple failures that occur simultaneously.

7.2.1 Managed System Description

The Trading System, introduced in Chapter 3, consists of the structural elements shown in Figure 7.1. These structural elements (at the runnable level) cooperate with each other to execute different use-cases.

![Figure 7.1: The structural elements of the Trading System.](image)

The Browser interacts with users. The AccessManager is responsible for verifying user credentials. The BusinessIntegrator is the core element of the system. It requests information from the DatabaseManager or the ServiceProvider, stores information in the database, generates HTML pages, and sends them to users. The ServiceProvider provides a number of Web Services that obtain their information from the BusinessManager (a legacy mainframe system). A prototype of the Trading System has been implemented.

7.2.2 Use-Case Descriptions

Before describing the use-cases, note that the Authentication Realisation use-case is triggered by an end-user action, and the Payment Status use-case is triggered by a timer within the system. The timer periodically initiates this use-case. The system executes the two use-cases simultaneously when an end-user requests access to the system while the payment status use-case is being executed. Both use-cases use the

---

6Each runnable is implemented as a Java RMI [175] server object, and runs inside its own Java Virtual Machine (JVM) [133]. In the real world, the runnables use various protocols to communicate with each other. For example, the Browser-AccessManager communication uses the HTTPS protocol and the BusinessIntegrator-ServiceProvider communication uses the SOAP protocol. However, for the sake of simplicity, the communication between all runnables, for the purpose of the case studies, is implemented using the Java Remote Method Invocation (JRMI) [175] protocol.
Illustrative Scenarios

structural elements DatabaseManager, BusinessIntegrator, ServiceProvider, and BusinessManager. The Authentication Realisation use-case also uses the Browser and AccessManager. As a result, the use-cases have many common structural elements.

Authentication Realisation Use-Case

The authentication realisation use-case describes the way users authenticate themselves in the secure Trading System. Users provide their certificates through the Browser to the AccessManager. Upon receiving a certificate, the AccessManager verifies its validity, and passes it to the BusinessIntegrator. The BusinessIntegrator consults the DatabaseManager to construct authentication information based on the user's identity specified in the certificate. The authentication information is then sent to the ServiceProvider. The ServiceProvider provides a number of Web Services that are also used by other systems of the company. Many of these Web Services access distributed resources. To this end, the ServiceProvider maintains a connection pool with the BusinessManager, as shown in Figure 7.2. One of the Web Services is the Authentication service that sends the authentication request to the BusinessManager.

![Figure 7.2: The process of sending authentication requests to the BusinessManager.](image)

The BusinessManager authenticates the user and returns the result back to the user's Browser through the BusinessIntegrator. The semi-formal use-case description of the Authentication Realisation use-case is depicted in Figure 7.3.

Payment Status Use-Case

To explain how the Payment Status use-case works, first the situation before execution of this use-case is briefly described. Users provide their bank account information to the Trading System and request payment of a selected trade. The BusinessIntegrator sends the payment order to the BusinessManager, and adds the selected trade to a list of trades of which the payment have been requested. The actual payment realisation, performed by the back office systems and bank employees, requires processing time. Users can consult the system to enquire the status of their payment after payment request. Initiating and maintaining a communication link with the back office systems and obtaining the status of payments is the purpose of the Payment Status use-case.
name: Authentication Realisation  
actors: User  
trigger: passing certificate  
pre-conditions: valid certificate  
post-conditions: user authenticated  
structural elements: Browser, AccessManager, BusinessIntegrator, ServiceProvider, BusinessManager  
steps:  
(1) Browser passes user’s certificate to AccessManager,  
(2) AccessManager passes certificate to BusinessIntegrator,  
(3) BusinessIntegrator calls ‘Authentication Preparation’ to obtain the auth-info,  
(4) BusinessIntegrator sends the auth-info to ServiceProvider,  
(5) ServiceProvider sends the auth-info to BusinessManager,  
(6) BusinessManager sends the auth-result to ServiceProvider,  
(7) ServiceProvider sends the auth-result to BusinessIntegrator,  
(8) BusinessIntegrator sends the auth-result to Browser.

Figure 7.3: Authentication Realisation use-case description at runnable level.

The status of a payment is one of the following: unknown, pending, rejected, or processed. Periodically (e.g., every 10 seconds), the BusinessIntegrator sends a request to the DatabaseManager to provide the list of trades of which the payment status is unknown or pending. The list is sent to the Payment service hosted by the ServiceProvider. For each item in the list, the Payment service sends a payment status request to the BusinessManager using the ServiceProvider’s connection pool, as shown in Figure 7.4.

As the BusinessManager can handle only a single request per connection and there are only a limited number of connections, the ServiceProvider puts the items of the list (of payment status requests) in an internal queue. As soon as a connection in the pool becomes free, the ServiceProvider chooses a request from the queue and sends it to the BusinessManager through that connection. The BusinessManager only responds to the request if the payment status of the related trade has changed. The semi-formal use-case description of the Payment Status use-case is depicted in Figure 7.5.
name: Payment Status  
actors: Timer  
trigger: Time expiration  
pream: Payment orders are in Database  
post-am: Payment status is updated  
structural elements: DatabaseManager, BusinessIntegrator, ServiceProvider, BusinessManager  
steps:  
(1) BusinessIntegrator receives ordered-payments (list) from DatabaseManager,  
(2) BusinessIntegrator passes the ordered-payments to ServiceProvider,  
(3) ServiceProvider sends the ordered-payments one by one to BusinessManager,  
(4) BusinessManager sends the status-upgrade to ServiceProvider,  
(5) ServiceProvider sends the status-upgrade to BusinessIntegrator,  
(6) BusinessIntegrator sends status-upgrades (list) to DatabaseManager,  
(7) DatabaseManager deletes ‘processed’ or ‘rejected’ payments from database.

Figure 7.5: Payment Status use-case description at runnable level.

7.2.3 Self-Management Model Specification

This section depicts the specification of the relevant part of the self-management model (relevant for self-diagnosis) for the use-cases described in Section 7.2.2. The OWL specifications of the use-cases (Jobs) are presented, the autonomic manager of each job and the corresponding symptoms and hypotheses are specified, and the SWRL rules with which to reason about these symptoms and hypotheses are sketched.

Managed System Model Specification

This section focuses on the behavioural model specification. A use-case is specified in OWL in terms of a job and tasks. The objective is to show how a job and a task are specified, how a sensor is related to a job and a task, and how a task is related to a managed structural element executing the task. The Authentication Realisation use-case is specified as AuthRealJob, and the Payment Status use-case is specified as PaymentStatusJob.

Figure 7.6: Job specification of the Authentication Realisation use-case in OWL.

Figure 7.6 depicts the OWL code for the Authentication Realisation use-case. The job has one input (JICertificate), one output (JOAuthResult), and a number of tasks (such as SendAuthInfo2BusinessManagerTask). The AuthRealJob-
7.2 Case 1: Single-Level Use-Case Management

Start and AuthRealJobEnd are sensors that indicate the start and end of the job. SendAuthInfo2BusinessManagerTask is a state interaction task (see Section 5.1.2), runs inside the structural element ServiceProvider, and is monitored by the TimeoutExceptionSensor. As depicted in Figure 7.7, this sensor monitors the occurrence of the TimeoutEvent and is instrumented in the byte-code of the SendAuthInfo2BusinessManager method of the ServiceProviderImpl Java class.

In addition to the jobs specified above, two administrative jobs (see Section 4.8.4) are introduced: (1) a system level job (AuthPaymentJob) to control the simultaneous execution of the AuthRealJob and PaymentStatusJob, and (2) a runnable level job (SpBmConnCheckJob) to inspect the connection between the ServiceProvider and BusinessManager. The tasks of the first job is to invoke both AuthRealJob and PaymentStatusJob. The tasks of the second job is to read the size of the connection pool and the current number of connections.

Autonomic Manager Specification

Four autonomic managers are specified for each of the jobs specified in the previous section: AuthPaymentAM, AuthRealAM, PaymentStatusAM, and SpBmConnCheckAM. Figure 7.8 depicts the OWL code of the AuthPaymentAM. The job that this autonomic manager manages is AuthPaymentJob. The autonomic manager has three children: AuthRealAM, PaymentStatusAM, and SpBmConnCheckAM that each deliver the result of their autonomic process to their parent. Furthermore, the figure depicts the symptoms (AuthRealisationFailed, StatusUpgradeFailed, BrokenSpBmConn, and SpBmCommonChannelBlocked) and hypothesis (LowSpBmConnPoolSize) regarding the job managed by AuthPaymentAM. The figure also depicts the autonomic process result (AuthPaymentAPR) of AuthPaymentAM. This autonomic process result contains the determined diagnosis of AuthPaymentAM.
Symptoms & Hypotheses Specification

Each autonomic manager has its own set of symptoms and hypotheses. The following describes the symptoms and hypotheses belonging to the knowledge-base of AuthPaymentAM, as depicted in Figure 7.9. The AuthRealisationFailed symptom indicates failure of the AuthRealJob, and the StatusUpgradeFailed symptom indicates failure of the PaymentStatusJob. The BrokenSpBmConn symptom indicates that the connection between the ServiceProvider and BusinessManager has failed (i.e., connection is not available). The SpBmCommonChannelBlocked symptom indicates that the connection is available but no data traffic is possible. To inspect the value of SpBmCommonChannelBlocked, an inspective plan (SpBmConnCheckInspectivePlan) is specified. The figure also depicts the OWL code for the LowSpBmConnPoolSize hypothesis and its relevant symptoms.

During runtime, the value of a symptom is determined by the values of one or more sensors, the autonomic process results of one or more children, and/or the result of an inspective plan. Figure 7.10 illustrates how the values of the relevant symptoms of the LowSpBmConnPoolSize hypothesis are determined. AuthRealisationFailed and StatusUpgradeFailed are respectively determined by
7.2 Case 1: Single-Level Use-Case Management

the autonomic process result of the AuthRealAM and PaymentStatusAM. The BrokenSpBmConn is determined by the value of the SpBmConnSensor, and the SpBmCommonChannelBlocked is determined by the autonomic process result of the SpBmConnCheckAM that manages the execution of the inspective plan.

Figure 7.10: The relation between specific hypotheses, symptoms, and inspective plans.

SWRL Rules Specification

As stated in Chapter 4, the framework provides a set of generic rules that are used by all specific models. The SWRL rule in Figure 7.11 is one of the generic rules used by the system level autonomic manager (AuthPaymentAM) to determine whether or not an inspective plan is to be activated. Recall that the autonomic process result of an autonomic manager includes the diagnosis determined. This diagnosis includes information about the task (the abnormal task) that has caused the failure. Also, recall that a state interaction task sends a state from one structural element (runnable) to another structural element (runnable), related to each other through a connector. The generic rule in Figure 7.11 indicates that an autonomic manager will perform an inspective action for the connection between two runnables if the autonomic manager receives diagnoses, determined by its children, that include abnormal tasks of the same type (SendStateInteraction), and the structural elements (runnables) associated with these tasks are the same.

Figure 7.11: The generic rule for inspection of a connector status.

This section presents a number of specific rules that reason about the relation between sensors, symptoms, and hypotheses belonging to the knowledge-base of
an autonomic manager. The SWRL rules in Figure 7.12 are part of the knowledge-base of the AuthRealAM. The first rule is a symptom occurrence rule, and indicates that the occurrence of the TimeoutExpOccurred symptom depends on the value of the TimeoutExceptionSensor. The second rule is a hypothesis selection rule, and confirms that the AuthRealisationFailure hypothesis must be selected if and when the TimeoutExpOccurred occurs.

\[
\text{observedValue(TimeoutExceptionSensor, 'occurred')} \rightarrow \text{arisen(TimeoutExpOccurred, Pos)}
\]
\[
\text{arisen(TimeoutExpOccurred, Pos)} \rightarrow \text{userDefSelCriteria(AuthRealisationFailure, true)}
\]

**Figure 7.12:** The specific analysis and diagnosis rules for the AuthRealAM.

The SWRL rule in Figure 7.13 is included in the knowledge-base of SpBmConnCheckAM. The value of the SpBmConnPoolSizeSensor specifies the maximum number of connection slots between the ServiceProvider and BusinessManager, and the value of the SpBmNumOfConnsSensor is the actual number of connection slots used. The SWRL rule indicates that the data traffic blocks (the SpBmConnBlocked symptom occurs) if the actual number of connection slots used equals the maximum number of connection slots.

\[
\text{observedValue(SpBmConnPoolSizeSensor, ?v1)} \land \text{observedValue(SpBmNumOfConnsSensor, ?v2)} \land \text{swrlb:equal(?v1, ?v2)} \rightarrow \text{arisen(SpBmConnBlocked, Pos)}
\]

**Figure 7.13:** The specific analysis rule for the SpBmConnCheckAM.

The SWRL rule in Figure 7.14 is included in the knowledge-base of AuthPaymentAM. This rule is a hypothesis evaluation rule, and states that all connection slots between the ServiceProvider and BusinessManager are in use, indicating that the pre-defined maximum number of connection slots (connection pool size) is too low. As a result, the data traffic is blocked between the ServiceProvider and BusinessManager.

\[
\text{arisen(BrokenSpBmConn, Neg)} \land \text{arisen(SpBmCommonChannelBlocked, Pos)} \rightarrow \text{diagnoser:userDefEvalCriteria(LowSpBmConnPoolSize, true)}
\]

**Figure 7.14:** The specific diagnosis rule for the AuthPaymentAM.

### 7.2.4 Simultaneous Failure Diagnosis

Note that use-cases, such as those described in the previous sections, require a number of runnables communicating with each other using diverse protocols. As a result, there are many places where things can go wrong during the execution of these use-cases. In the case study presented in the previous sections, two failures are assumed to occur simultaneously. The first failure is that the Trading System denies to grant access to the user, even though he/she has a valid certificate and identity. The second failure is that the Trading System does not confirm the payment status of one or more trades, even though the employees of the back
office have assured the user over other channels (e.g., fax, phone) that his/her payment has been successfully processed. The possible root-causes of these failures are that the number of trades of which the payment status is unknown or pending increases to exceed the size of the connection pool, and the system repeatedly requests authenticating a user. The following explains how the root-causes of these multiple failures are found by the autonomic managers.

After instrumenting sensors in the code of the Trading System and generating autonomic management, the autonomic manager at the highest hierarchical level is started. As shown in Figure 7.15 (the log of execution), this autonomic manager loads the OWL concepts and individuals specified in the OWL file (line (1)) into the execution environment. Immediately thereafter, the rule engine is instantiated to prepare it for interpreting the SWRL rules (line (2)). Line (3) shows that a communication channel between autonomic management and the messaging software is set up to consume messages. Lines (4) through (7) show that each autonomic manager is responsible for creating and starting its children, job, analyser, and diagnoser. After these initialisation activities, all autonomic managers wait for sensor values from the Trading System to arrive.

![Figure 7.15: Part of the log file showing the initialisation activities.](image)

Figure 7.15 shows that the PaymentStatusAM receives the job start indicator, the timeout exception, and job end indicator.

![Figure 7.16: Part of the log file showing the arrival of sensor values.](image)

As soon as the job end indicator arrives, the PaymentStatusAM starts its autonomic process. Figure 7.17 depicts a trace of the autonomic process. Before executing the symptom occurrence rules, the value of the TimeoutExceptionSensor is stored in the OWL model to enable the rule engine to execute the related SWRL rule (line (3)). Line (5) shows the inferred axiom that underlies the occurrence of the TimeoutExpOccurred. Lines (6) through (12) show the diagnostic process. Line (13) shows that the diagnostic process starts again to see whether there are any hypotheses to reason about. Because the StatusUpgradeFailure has already been evaluated, the diagnostic process enters its last phase by performing the diagnosis determination rules (lines (15) and (16)). So, the PaymentStatusAM determines the failure of the payment status upgrade. After diagnosis determina-
tion, the autonomic process result is delivered to the parent autonomic manager (line (17)).

(1) Starting autonomic process for AM 'PaymentStatusAM'...
(2) Performing Analysis Rules for AM 'PaymentStatusAM'...
(3) Value 'occurred' of sensor 'TimeoutExceptionSensor' stored.
(4) Performing Analysis Symptom Occurrence Rules ...
(5) InferredAxiom: arisen(TimeoutExpOccurred, Pos)
(6) Performing Diagnosis Rules for AM 'PaymentStatusAM'...
(7) Performing Hypothesis Selection Rules ...
(8) InferredAxiom: focussed(StatusUpgradeFailure, true)
(9) Performing Hypothesis Validation Rules ...
(10) InferredAxiom: validated(StatusUpgradeFailure, Pos)
(11) Performing Diagnosis Determination Rules ...
(12) InferredAxiom: evaluted(StatusUpgradeFailure, true), assessed(StatusUpgradeFailure, true)
(13) Performing Hypothesis Selection Rules ...
(14) InferredAxiom: tried(StatusUpgradeFailure, true), focussed(StatusUpgradeFailure, false)
(15) Performing Diagnosis Determination Rules ...
(16) InferredAxiom: determined(StatusUpgradeFailure, true)
(17) childResult 'PaymentStatusAPR' received for 'AuthPaymentAM'.

Figure 7.17: Part of the log file showing the autonomic process of the PaymentStatusAM.

As soon as the execution of the Authentication Realisation use-case finishes, the AuthRealAM starts its autonomic process as shown in Figure 7.18. Line (3) shows that the AuthRealAM determines the failure of the authentication process. Finally, the autonomic process result of the AuthRealAM is delivered to the same parent autonomic manager as the PaymentStatusAM (line (4)).

(1) Starting autonomic process for AM 'AuthRealAM'...
(2) Performing Diagnosis Determination Rules ...
(3) InferredAxiom: determined(AuthRealisationFailure,true)
(4) childResult 'AuthRealAPR' received for 'AuthPaymentAM'.

Figure 7.18: Part of the log file showing the autonomic process of the AuthRealAM.

The autonomic process of the parent (AuthPaymentAM) starts when its job end indicator arrives. As shown in line (3) of Figure 7.19, it is confirmed that the BrokenSpBmConn symptom has not occurred. However, the occurrence of the SpBmCommonChannelBlocked is unknown. Therefore, the LowSpBmConnPoolSize hypothesis is not considered to be valid while performing the hypothesis validation rules (line (8)). That is the reason why no diagnosis is determined (line (11)). As a result of the execution of the generic validation rule, shown in Figure 7.11, the SpBmConnCheckInspectivePlan is started (lines (9) and (10)) to inspect the occurrence of the unknown symptom.

The autonomic manager that manages the execution of the inspective plan is SpBmConnCheckAM which is the child of the AuthPaymentAM. After the termination of the inspective plan, the SpBmConnCheckAM delivers its autonomic process result
(1) Starting autonomic process for AM 'AuthPaymentAM'...
(2) Performing Analysis Rules for AM 'AuthPaymentAM'...
(3) Inferred Axiom: arisen(BrokenSpBmConn, Neg)
(4) Performing Hypothesis Selection Rules ...
(5) Inferred Axiom: focussed(LowSpBmConnPoolSize, true)
(6) Unknown symptoms: 'SpBmCommonChannelBlocked'
(7) Performing Hypothesis Validation Rules ...
(8) Inferred Axiom: validated(LowSpBmConnPoolSize, Neg)
(9) Inspective Plan 'SpBmConnCheckInspectivePlan', associated with the symptom 'SpBmCommonChannelBlocked', is going to be executed...
(10) Executing command: cmd.exe /C "cd classes&&java -classpath ... SpBmConnCheckConst"
(11) Performing Diagnosis Determination Rules ...
(12) Inferred Axiom:

Figure 7.19: Part of the log file showing the autonomic process of the AuthPaymentAM.

Finally, the AuthPaymentAM determines that the size of the connection pool is low. This diagnosis, which is based on the information provided by the PaymentStatusAM, AuthRealAM, and SpBmConnCheckAM, explains the failure of both payment status upgrade and authentication process.

(1) Starting autonomic process for AM 'AuthPaymentAM'...
(2) Performing Hypothesis Selection Rules ...
(3) Inferred Axiom: focussed(LowSpBmConnPoolSize, true)
(4) Performing Hypothesis Validation Rules ...
(5) Inferred Axiom: validated(LowSpBmConnPoolSize, Pos)
(6) Performing Hypothesis Evaluation Rules ...
(7) Inferred Axiom: evaluated(LowSpBmConnPoolSize, true), assessed(LowSpBmConnPoolSize, true)
(8) Performing Diagnosis Determination Rules ...
(9) Inferred Axiom: determined(LowSpBmConnPoolSize, true)

Figure 7.20: Part of the log file showing the autonomic process of the AuthPaymentAM after performing the inspective plan.

The above case study shows that an autonomic manager is able to combine generic SWRL rules, provided by the framework, and specific SWRL rules, provided by domain experts, to infer the root-cause of multiple failures occurring simultaneously. In addition, the trace shows that if a certain symptom is unknown, the autonomic manager is able to execute the pre-defined inspective plan, observe the real world, and incorporate this observation in its diagnostic process.

7.3 Case 2: Multi-Level Use-Case Management

Pinpointing the root-cause of system failures in complex systems is often a challenge. The framework proposed in this thesis provides an explicit means with which to structure use-cases and a means to define relations between use-cases (see Section 3.5 for the description of use-case levels). Autonomic managers of
use-cases, and their relations are structured in the same way (see Section 4.7.2). This section shows how autonomic managers at different levels cooperate with each other to pinpoint the root-cause of a problem.

7.3.1 Managed System Description

The use-cases in this case study execute on a number of components and classes of the runnables BusinessIntegrator, DatabaseManager, and ServiceProvider of the Trading System. As shown in Figure 7.21, three components of the BusinessIntegrator and two classes within one of the components, one component of the DatabaseManager and one of its classes, and one component of the ServiceProvider are deployed. The following sections describe how these structural elements cooperate to execute the various use-cases.

![Diagram of the structural elements of the Trading System at multiple levels.](Figure 7.21)

7.3.2 Use-Case Descriptions

To illustrate multi-level diagnosis, this section describes six use-cases organised at different use-case levels (see Figure 7.22). The top level use-case (Authentication Preparation) invokes three component level use-cases: Certificate Parsing, User-info Update, and Auth-info Preparation. In turn, the last component level use-case invokes two class level use-cases: User-record Retrieval and Auth-info Construction.

Authentication Preparation use-case

The goal of the Authentication Preparation use-case is to prepare authentication information based on the provided certificate. The user’s identity (userid) is extracted from the certificate, the corresponding user information is obtained from the mainframe (user-info), and is stored in the database (user-record) after enhancement. The user-record is used to construct the authentication information (auth-info). The actual work is done by the three component level use-cases. The semi-formal use-case description of the Authentication Preparation use-case is depicted in Figure 7.23.
### Certificate Parsing Use-Case

The **Certificate Parsing** use-case is responsible for parsing the certificate, extracting the user’s identity, and passing it to its parent use-case. Usually, the user’s identity in a company is based on a specific pattern (e.g., `firstname.lastname`). The assumption is that a user’s identity follows that pattern in all certificates. If the use-case encounters a mismatch, it stops processing. The semi-formal use-case description of the **Certificate Parsing** use-case is depicted in Figure 7.24.

```plaintext
name: Certificate Parsing
actors: BusinessIntegrator sub-system
trigger: passing certificate
pre-conditions: valid certificate
post-conditions: user’s identity
structural elements: CertificateParserComp, UserInfoMaintainerComp
steps:
(1) CertificateParserComp extracts userid,
(2) CertificateParserComp passes userid to UserInfoMaintainerComp.
```

**Figure 7.24:** Certificate Parsing use-case at component level.
User-info Update Use-Case

The User-info Update use-case is responsible for obtaining user information (such as password, department, role, country, etc.) from the back-end sub-system and storing the information in a local database. The use-case first queries the database to see whether the given user is known. Otherwise, it makes a connection with a Web Service to obtain the information from the back-end sub-system. The semi-formal use-case description of the User-info Update use-case is depicted in Figure 7.25.

![Figure 7.25: User-info Update use-case at component level.](image)

Auth-info Preparation Use-Case

The Auth-info Preparation use-case is responsible for retrieving the user record from the database and constructing authentication information. This component level use-case utilises two class level use-cases for this purpose: User-record Retrieval and Auth-info Construction. The semi-formal use-case description of the Auth-info Preparation use-case is depicted in Figure 7.26.

![Figure 7.26: Auth-info Preparation use-case at component level.](image)

User-record Retrieval Use-Case

The User-record Retrieval use-case is responsible for retrieving user records from the database. The semi-formal use-case description of this use-case is depicted in Figure 7.27.
7.3 Case 2: Multi-Level Use-Case Management

name: User-record Retrieval
actors: PrepareAuthComp component
trigger: passing user’s identity
pre-conditions: valid user’s identity, user-record in database
post-conditions: user-record retrieved
structural elements: UserRecordRetrieveClass, DBTableResponseClass
steps:
(1) UserRecordRetrieveClass passes the userid to DBTableResponseClass,
(2) UserRecordRetrieveClass receives the user-record from DBTableResponseClass.

Figure 7.27: User-record Retrieval use-case at class level.

Auth-info Construction Use-Case

The back-end sub-system, responsible for authenticating users and initiating a session for them, requires authentication information consisting of a user’s userid, password, role, and country. The Auth-info Construction use-case checks the values of the mentioned fields to see whether they have been specified or not. If so, it constructs the auth-info object and initiates the connection with the Authentication Web Service. The semi-formal use-case description of the Auth-info Construction use-case is depicted in Figure 7.28.

name: Auth-info Construction
actors: PrepareAuthComp component
trigger: passing user-record
pre-conditions: valid user-record
post-conditions: auth-info constructed
structural elements: UserRecordRetrieveClass, AuthInfoConstructClass
steps:
(1) AuthInfoConstructClass receives the user-record from UserRecordRetrieveClass,
(2) AuthInfoConstructClass constructs the auth-info.

Figure 7.28: Auth-info Construction use-case at class level.

7.3.3 Self-Management Model Specification

This section describes how the model of the managed system and the relation between autonomic managers at different levels of the hierarchy are specified in OWL. Also, the SWRL rules for multi-level diagnosis are specified.

Managed System Model Specification

This section focuses on the relation between the behavioural and structural model specification. Figure 7.29 depicts the OWL code of one of the tasks of the runnable level job AuthPrepJob. The task is an invocation of the component level job AuthInfoPrepJob, and it is executed inside the managed runnable BusinessIntegrator. This managed runnable consists of a number of connectors and components. The figure depicts only one of its connectors (BiDmConnector) and one of its components (PrepareAuthComp). The BiDmConnector connects the BusinessIntegrator to the DatabaseManager using the JDBC protocol.
As depicted in Figure 7.30, the AuthInfoPrepJob has two invocation tasks. The UserRecordRetrieveInvokeTask invokes the class level job UserRecRetJob, and the AuthInfoConstInvokeTask invokes the class level job AuthInfoConsJob. Both tasks are executed inside the managed component PrepareAuthComp. This managed component consists of two classes: UserRecordRetrieveClass and AuthInfoConstructClass.

Figure 7.31 depicts one of the tasks (SendUserId2DBClassTask) of the class level job UserRecRetJob. The SendUserId2DBClassTask is an interaction task between the two managed classes UserRecordRetrieveClass and DBTableResponseClass. The first class belongs to the managed component PrepareAuthComp.
AuthComp and the second class belongs to one of the components of the Database-Manager.

```xml
<behav-model:ClassLevelJob rdf:ID="UserRecRetJob">
  <behav-model:tasks rdf:resource="#SendUserId2DBClassTask"/>
</behav-model:ClassLevelJob>
<behav-model:SendStateInteraction rdf:ID="SendUserId2DBClassTask">
  <behav-model:managedStructuralElement rdf:resource="#UserRecordRetrieveClass"/>
  <behav-model:partnerManagedStructuralElement rdf:resource="#DBTableResponseClass"/>
  ...
</behav-model:SendStateInteraction>
<behav-model:ClassLevelJob rdf:ID="AuthInfoConsJob">
  <behav-model:tasks rdf:resource="#ConstructAuthInfoTask"/>
  ...
</behav-model:ClassLevelJob>
<behav-model:ReturnJobOutput rdf:ID="ConstructAuthInfoTask">
  <behav-model:managedStructuralElement rdf:resource="#AuthInfoConstructClass"/>
  ...
</behav-model:ReturnJobOutput>
<struct-model:AtomicManagedRunnable rdf:ID="DatabaseManager">
  <struct-model:connector rdf:resource="#BiDmConnector"/>
  <struct-model:subElement rdf:resource="#DBTableMaintainerComp"/>
</struct-model:AtomicManagedRunnable>
<struct-model:AtomicManagedComponent rdf:ID="DBTableMaintainerComp">
  <struct-model:subElement rdf:resource="#DBTableResponseClass"/>
</struct-model:AtomicManagedComponent>
<struct-model:AtomicManagedClass rdf:ID="DBTableResponseClass"/>
```

Figure 7.31: The specification of the UserRecRetJob and AuthInfoConsJob including their tasks and associated structural elements in OWL.

**Autonomic Manager Specification**

To determine the root-cause of abnormal behaviour at different levels, seven autonomic managers are specified: one per use-case described in the previous section and one for the management of the inspective plan used to inspect the status of the connector between the BusinessIntegrator and DatabaseManager. Figure 7.32 shows the relationship between the autonomic managers and their private knowledge (symptoms and hypotheses).

The OWL code for each autonomic manager is straightforward. In Figure 7.33, only the OWL code for the AuthInfoConsAM is depicted. This autonomic manager communicates its result to its parent, AuthInfoPrepAM, through AuthInfoConsAPR. It has a number of sensors for monitoring its job, AuthInfoConsJob, and a number of symptoms, hypotheses, and SWRL rules that are explained in the coming sections.

**Symptoms & Hypotheses Specification**

Different abnormal behaviours can occur at each level of the hierarchy of the use-cases (see Figure 7.32). For example, ParseError can occur when parsing the certificate by the CertParseJob, the retrieval of user information by the User-InfoUpdateJob from the ServiceProvider can fail (UserInfoRetrieveFailed), ob-
Figure 7.32: The relationships among the autonomic managers of the use-cases described before. The knowledge of each autonomic manager regarding the abnormal behaviour of its managed use-case and the possible root-causes are depicted. An arrow shows the flow of autonomic management information from a child to its parent.

```xml
<autonomic-manager:AutonomicManager rdf:ID="AuthInfoConsAM">
  <autonomic-manager:job rdf:resource="#AuthInfoConsJob"/>
  <autonomic-manager:autonomicProcessResult rdf:resource="#AuthInfoConsAPR"/>
  <autonomic-manager:hypotheses rdf:resource="#EmptyUserRecord"/>
  <autonomic-manager:hypotheses rdf:resource="#EmptyPasswordField"/>
  <autonomic-manager:parentAutonomicManagers rdf:resource="#AuthInfoPrepAM"/>
  <autonomic-manager:sensors rdf:resource="#AuthExpSensor"/>
  <autonomic-manager:sensors rdf:resource="#AuthInfoConsJobEnd"/>
  <autonomic-manager:sensors rdf:resource="#AuthInfoConsJobStart"/>
  <autonomic-manager:symptoms rdf:resource="#TSPasswordSensor"/>
  <autonomic-manager:symptoms rdf:resource="#PasswordNotSpecified"/>
  ...
</autonomic-manager:AutonomicManager>
```

Figure 7.33: The specification of the AuthInfoConsAM autonomic manager in OWL.

Obtaining the user record by the UserRecRetJob from the DatabaseManager can be unsuccessful (UserRecordRetrieveFailed), or an exception can occur (AuthExpOccurred) during construction of the authentication information by the AuthInfoConsJob. Here, in Figure 7.34, only the OWL code of the hypotheses concerning the AuthInfoPrepJob and AuthInfoConsJob is depicted.

Note that the importance (weight) of the hypothesis UserRecRetFailure is higher than the AuthInfoConsFailure, because the output of the UserRecRetJob is the input for the AuthInfoConsJob. The authentication information can
be constructed only if the user record is retrieved from the local database. Also note that the hypothesis \textit{EmptyPasswordField} is the sub-hypothesis of the \textit{EmptyUserRecord}, because the password field is part of the user record. In the following section, the generic SWRL rules use this information to determine the proper diagnosis.

**SWRL Rules Specification**

The two generic SWRL rules that impact the determination of hypotheses are depicted in Figure 7.35. The first rule states that if the evaluated set of hypotheses contains more than one hypothesis then the pre-defined weights of the hypotheses are compared and the one with the highest weight is chosen as the final diagnosis. The second rule expresses a preference for a parent hypothesis (more generic root-cause) instead of a child hypothesis (more specific root-cause). Based on these rules, the determined diagnosis of the \textit{AuthInfoPrepAM} is the \textit{UserRecRetFailure} instead of \textit{AuthInfoConsFailure}, and the determined diagnosis of the \textit{AuthInfoConsAM} is the \textit{EmptyUserRecord} instead of \textit{EmptyPasswordField}.

**Figure 7.35:** The generic rule for choosing the hypotheses that are more important and represent more generic root-causes.

Figure 7.36 depicts part of the specific rules for the \textit{AuthPrepAM}. Both rules indicate that the retrieval of the user record from the database was not successful.
The first rule concludes that the root-cause of this symptom is the fact that the \textit{DatabaseManager} is not available because the database does not accept any connections. However, the second rule concludes that the connection to the \textit{DatabaseManager} is not properly configured.

\[
\begin{align*}
arisen(\text{UserRecordRetrievalFailed}, \text{Pos}) \land & \arisen(\text{DBConnFailed}, \text{Pos}) \land \\
\rightarrow & \ \text{userDefEvalCriteria}(\text{DBNotAvailable}, \text{true}) \\
\arisen(\text{UserRecordRetrievalFailed}, \text{Pos}) \land & \arisen(\text{DBConnFailed}, \text{Neg}) \land \\
\rightarrow & \ \text{userDefEvalCriteria}(\text{ImproperDBConfig}, \text{true})
\end{align*}
\]

\textbf{Figure 7.36:} The specific diagnosis rules for the AuthPrepAM.

\subsection{7.3.4 Multi-Level Failure Diagnosis}

In this case study, the \textit{DatabaseManager} goes down and thereafter a user initiates an access request. Consequently, the \textit{Trading System} is not able to prepare the authentication information. The unavailability of the \textit{DatabaseManager} causes the failure of the \textit{User-info Update} use-case to store the retrieved user information into the database. The following explains how this root-cause of the failure is found by the autonomic management.

As in the first case study, after starting autonomic management, the \textit{Trading System} is started. The first part of the log file, logged by the autonomic management, is more or less the same as the log file of the first case study (see Figure 7.15). In the following, the traces of the autonomic processes of the AuthInfoConsAM and AuthPrepAM are described.

The autonomic process of the AuthInfoConsAM is started when its job end indicator arrives (see Figure 7.37). The autonomic manager receives the values of both \textit{AuthExpSensor} and \textit{TSPasswordSensor} and stores their values in the OWL model (lines (3) and (4)) to reason about its symptoms. Line (6) shows the inferred symptoms. The interesting part is line (15). Although both \textit{EmptyUserRecord} and \textit{EmptyPasswordField} hypotheses are evaluated (line (13)), \textit{EmptyUserRecord} hypothesis is determined to be the root-cause. That is because of the execution of the generic SWRL rule depicted in Figure 7.35. After diagnosis determination, the autonomic process result is delivered to the parent autonomic manager (line (16)) which causes the execution of the mapping rules. These rules infer the occurrence of the parent’s symptom (\textit{EmptyAuthInfo}) (line (18)).

Figure 7.38 shows the autonomic process of the top level autonomic manager, namely AuthPrepAM. Two hypotheses (from six hypotheses) are selected (line (4)) to investigate whether they are the root-causes but only one of them (\textit{DBNotAvailable}) is determined to be the diagnosis (line (10)). That is because of the execution of the \textit{hypothesis evaluation rules} depicted in Figure 7.36.

The above case study shows that a failure in one part of a distributed system can result in different abnormal behaviours in other parts of the system. To diagnose such a failure, a complex use-case is divided into several use-cases at different
levels. This division provides a suitable basis for relating the autonomic managers to each other in a hierarchical fashion. In this way, each autonomic manager in the hierarchy, on the one hand, is aware of the abnormal behaviour of only its own scope, and on the other hand, influences the diagnostic decision of its parent (i.e., making the multi-level diagnosis possible) by reporting its autonomic management result to its parent.

7.4 Concluding Remarks

In Section 1.4, a number of requirements were formulated regarding self-management of existing distributed systems. Furthermore, in the previous chapters, the design of the self-management framework for existing distributed systems was explained. Based on this design, two experiments were implemented. The following discusses how far the proposed self-management framework meets the requirements mentioned in Section 1.4.

The requirement of efficiently capturing system knowledge and integrating
views of domain experts is to a large extent satisfied. The framework provides an efficient way to capture system knowledge because it reuses the knowledge available in existing use-cases. The framework also provides a mechanism to categorise use-cases based on the views of domain experts. All these different use-case categories are finally converted into a single concept (Job) in the framework. The integration between different views of domain experts is accomplished by relating different instances of the single concept to each other. However, existing use-cases should be slightly adapted to be used by the framework. As use-cases are expressed in a semi-formal format, they have to be converted into the formal format required by the framework. The framework does not provide a mechanism to automatically convert the semi-formal notation into the formal one. This is a limitation of the framework.

The requirement of detecting, diagnosing, and repairing the most frequently occurring system malfunctions is partially satisfied. Analysers are responsible for detecting system malfunctions based on sensor values. The quality of detection depends on the type of sensors. The framework improves the quality of detection by providing a considerable number of sensor types which monitor various aspects of distributed systems. The framework also improves the quality of diagnosis by performing pro-active observations, and by combining the diagnostic results of multiple diagnosers. Although the framework provides basic concepts for planning, plan translation, and effectors, it cannot yet be used to repair a system malfunctioning. This is left as future work.

The requirement of being less intrusive on existing systems and domain experts is generally satisfied. The framework relieves system developers from writing additional system management code by automatically instrumenting sensors and effectors in the code of a managed system, and generating code for autonomic management. To minimise the impact of autonomic management on managed system, the framework let them execute in their own process spaces and communicate remotely with each other. Note that domain experts should still provide the code for specific sensors and effectors. The framework is unable to provide such codes.

7.5 Summary

This chapter depicts how the self-management framework is applied to management of examples of real-life problems. Two case studies are used to illustrate different aspects of the framework.

The first case study concerns the diagnosis of the simultaneous occurrence of two failures. The experiment shows that the autonomic management quickly becomes aware of the occurrence of both failures regarding two use-cases, and suggests the possible root-cause of the failures by utilising a generic diagnostic rule.

The second case study illustrates the ability of the framework to find the root-cause of a problem regarding a complex use-case. The use-case is divided into a hierarchy of simpler use-cases, and their autonomic managers are related to each other accordingly, making a hierarchy of parent-child relationships. The diagnostic
process of an autonomic manager is influenced by the diagnostic results of its children.
Chapter 8

The Execution Environment

The core component of the self-management framework presented in this thesis is the self-management model. The self-management model contains management and system specifications in OWL and SWRL (respectively presented in Chapters 4 and 5). These specifications themselves are not executable. To perform autonomic management of a distributed system, an execution environment is needed. This chapter introduces the execution environment, designed and implemented as part of the framework. The execution environment has been implemented in Java.

8.1 Execution Environment Overview

The execution environment consists of the following main components depicted in Figure 8.1: activation & control engine and rule engine. These components interact with self-management model and instrumented managed system.

![Figure 8.1: The components of the execution environment.](image)

The activation & control engine implements the autonomic process model described in Section 4.7.1 and depicted in Figure 4.12. This engine contains code with which to perform analysis, diagnosis, planning, and plan translation processes by an autonomic manager, to handle communication between an autonomic manager
and a managed system, to handle communication between autonomic managers, to invoke an inspective plan, and to handle communication between an autonomic manager and the rule engine. To accomplish these tasks, the activation & control engine utilises the knowledge on self-management concepts and their relationships specified in the self-management model. As mentioned in Section 6.4, the self-management model, specified in OWL and SWRL, consists of two parts: a generic part and a domain-specific part. The generic part is provided by the framework, and the domain-specific part is provided by domain experts.

The rule engine executes analysis, diagnostic, planning, and plan translation rules specified in SWRL. These rules are specified in the self-management model. The execution of rules is activated by the activation & control engine. The rule engine continuously applies a set of rules to a set of facts. The set of rules, loaded from the self-management model, is stored in the rule memory, and the set of facts, obtained from the activation & control engine, is stored in the fact memory. By applying rules to facts, the rule engine infers new facts that are passed to the activation & control engine. The activation & control engine stores the new facts in the self-management model.

The instrumented managed system is the executable code of a managed system extended with sensors and effectors. The activation & control engine obtains runtime information from sensors about the status of managed use-cases, and sends adaptation instructions to managed use-cases by activating effectors. The following sections explain in more detail the different aspects of the activation & control engine, the rule engine, and the instrumentation aspect of the instrumented managed system.

8.2 Activation & Control Engine

The code\textsuperscript{1} of the activation & control engine consists of two parts: generic code and domain-specific code. The generic code is provided by the framework. The generic code implements common control features of the autonomic process, and is used for the management of all managed systems. The domain-specific code is generated by the code generation tool of the framework based on the domain-specific part of the self-management model (see Figure 8.2). The domain-specific code implements the domain-specific control features, and is used for the management of a specific managed system.

\textbf{Figure 8.2:} The generation of code for an activation & control engine performed by the code generation tool of the self-management framework.

\textsuperscript{1}The complete code of the activation & control engine can be downloaded from \url{http://www.iids.org/research/self-management/self-management-framework.zip}. 

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{activation_control_engine.png}
\caption{The generation of code for an activation & control engine performed by the code generation tool of the self-management framework.}
\end{figure}
8.2 Activation & Control Engine

The code generation tool of the framework generates six Java packages. The packages contain the Java classes generated for the specified jobs and tasks, analysers and symptoms, diagnosers and hypotheses, autonomic managers and their autonomic management results, planners and inspective plans, and sensors and sensor value providers. The generic code (Java classes) together with the generated Java classes comprise the activation & control engine (i.e., autonomic management) for a specific managed system.

To completely separate the activities of autonomic management and its managed system, a design decision has been made to let the managed system and autonomic management execute in their own process spaces, and communicate remotely with each other. As a result, unexpected problems regarding autonomic management do not affect the normal working of the managed system. All autonomic managers run within one process space (the autonomic management process space). Each autonomic manager has its own thread of control within the autonomic management process space. The following sections explain some important features in the code of the activation & control engine (see Appendix B for parts of the Java code for these features).

8.2.1 Communication between Autonomic Management & Managed System

Communication between autonomic management and the sensors/effectors instrumented in a managed system has been implemented using the Apache ActiveMQ [9] messaging software. Apache ActiveMQ provides the publish/subscribe mechanism through which publishers send their messages to multiple subscribers. Utilising this mechanism, sensors publish their values to autonomic managers that have subscribed themselves to receive the value of a specific sensor. Furthermore, effectors have subscribed themselves to receive adaptation instructions which are published by autonomic managers.

To enable communication between autonomic management and an instrumented managed system, the different software elements need to be activated in an appropriate order. First, the messaging software is started. Second, autonomic management is started to subscribe to messaging software for consuming sensor values. Finally, the instrumented managed system is started to subscribe to messaging software for consuming adaptation instructions.

After an instrumented managed system is started, the instrumented sensors in the managed system send sensor values to autonomic management. An autonomic manager that receives a sensor value examines the type of the sensor value. The sensor value can indicate the start and end of a use-case, a state change, or an event occurrence within a use-case. Based on a sensor value, the autonomic manager


3The messaging software uses message topics to manage message flow from multiple publishers to multiple subscribers.
performs an appropriate action. A state change or an event occurrence causes the invocation of the analyser, diagnoser, planner, and plan translator of the autonomic manager. If there is a need to activate certain effectors then a message is sent to those effectors to become active.

### 8.2.2 Synchronisation between Autonomic Management & Managed System

When an autonomic manager receives a sensor value it spends time to analyse the current situation. The question that can be raised is: should the managed system wait for the response of the autonomic manager? The answer to this question is not straightforward and depends on the type of the managed system.

A ‘yes’ answer means that autonomic managers can run in sync with the managed system and are on time to perform remedy actions. However, forcing a managed system (especially a real-time system) to wait for the response of an autonomic manager can have undesirable affect on the behaviour of a managed system. That is because the communicating parties within a managed system have a certain expectation of each other’s response time. The response time of the communicating parties increases because of the overhead of autonomic managers. Therefore, a sequence of time-outs can occur.

A ‘no’ answer means that an autonomic manager always lags behind its managed system. The reason is that it takes time before a sensor value reaches an autonomic manager. The sensor values sent from the managed system are kept in the queue of autonomic management. In this case, autonomic managers are able to perform delayed problem determinations but it is hard for them to perform remedy actions because the managed system is not in the required state any more.

The default policy of the framework is based upon the ‘no’ answer: the managed system does not wait for the response of the autonomic manager. The motivation for this default policy is to minimise the influence of autonomic management on the regular working of the managed system.

### 8.2.3 Communication between Autonomic Managers

As described in Section 4.7.2, a distributed system’s behaviour is considered to be a collection of use-cases that are related to each other at various levels according to their references. Associating a separate autonomic manager to each use-case for managing its execution implies that autonomic managers should communicate and cooperate with each other to manage their use-cases. How autonomic managers communicate with each other is as follows.

As soon as an autonomic manager is started it creates all its children, assigns itself as their parent, and starts them one by one. Then, it starts one thread per child to wait for the autonomic process result of that child. When the autonomic process of a child autonomic manager finishes it sends its result to its parent.

For the sake of simplicity, communication between autonomic managers is implemented using the Java observer and observable pattern. According to the Java
observer and observable pattern [62], a Java object (observable) maintains a list of its dependents (observers) and notifies them automatically of any state changes. An alternative would have been to implement this communication using a messaging system such as the Apache ActiveMQ allowing remote communication between autonomic managers distributed across a network.

8.2.4 Invocation of Inspective Plan

After the hypothesis validation rules of an autonomic manager has been executed, the autonomic manager investigates the inferred facts to see whether the occurrence of a symptom should be inspected. If so, the autonomic manager reads the symptom specification from the OWL model, extracts its associated inspective plan, and invokes the plan's execute method. This method extracts the plan's ActionsConstruct class, prepares its runtime environment, and starts the class in a separate process. The ActionsConstruct is a Java class that invokes a set of atomic actions according to the specified construction scheme (see Section 4.8.4 for the description of different construction schemes such as Sequence, Parallel, Any-Order).

Each atomic action is a Java class that performs an inspective job such as determining the actual number of slots used in the connection pool between two runnables or inspecting a log file for the occurrence of an exception. This Java class, similar to the managed system’s code, is instrumented with sensors that inform the autonomic manager about the inspection results. Note that these Java classes are automatically generated by the framework.

8.2.5 Bootstrapping Autonomic Management

As stated above, autonomic managers are started by their parents. However, the system level autonomic managers responsible for the system level use-cases, introduced in Section 3.5, do not have a parent. The framework provides MasterAM and MasterJob for this purpose. The MasterAM is the parent of all system level autonomic managers, and it is responsible for managing MasterJob. The MasterJob contains tasks that are not representations of tasks in the managed system. They are administrative tasks, for example, a task that checks the correct execution order of the different runnables of the managed system.

The bootstrap process starts by manually starting the MasterAM. After the MasterAM is started it automatically starts the rule engine and all the system level autonomic managers, waiting for their results just the same as the regular autonomic managers.

8.3 Rule Engine

The different SWRL rules, such as the symptom occurrence rules or the hypothesis selection, validation, evaluation, and determination rules, specified in the generic
and domain models, are executed by the rule engine. When the rule engine\textsuperscript{4} completes the inference process, the inferred facts are transformed back into OWL knowledge.

Autonomic managers communicate with the rule engine as follows. When an autonomic manager is notified about a sensor, it stores the sensor’s observed value in the OWL model, reads its symptom occurrence rules from the model, and delivers them to the rule engine. The rule engine executes the rules and infers the occurrence of certain symptoms. Thereafter, the autonomic manager stores the inferred facts in the OWL model, and starts the diagnostic process. The result of the execution of the hypothesis determination rules (diagnosis) is stored as the autonomic process result of the autonomic manager in the OWL model. Finally, the autonomic manager reads its process result from the OWL model and sends it to its parent.

8.4 Instrumented Managed System

The instrumented managed system is an existing system extended with sensors and effectors. The required sensors and effectors can be instrumented in the code of an existing system manually or automatically. The framework offers an instrumentation tool to automatically instrument sensors and effectors in a managed system. This tool utilises the domain-specific part of the self-management model (see Figure 8.3). This part includes information about the instrumentation location (in the code of a managed system) specified by domain experts.

![Figure 8.3: The instrumentation of sensors and effectors in the code of a managed system performed by the instrumentation tool of the self-management framework.](image)

Using the instrumentation technique described in the following section, the instrumentation tool of the framework is able to perform automated instrumentation of a managed system written in an object-oriented language (e.g., Java, C\#, C++)\textsuperscript{5}.

\textsuperscript{4}This thesis uses the Jess [70] rule engine. As this rule engine is not familiar with the SWRL syntax, the SWRL-Jess bridge [142] is used to convert SWRL rules to Jess facts, templates, and slots.

\textsuperscript{5}The framework does not support the automated instrumentation of a managed system written in other programming languages (e.g., C, Pascal, Cobol). Note that the framework is still able to generate code for autonomic management for such a managed system if the required sensors
Note that a managed system can be written in multiple programming languages including object-oriented and other programming languages. In this case, only sensors and effectors belonging to the object-oriented part of the managed system are automatically instrumented. The sensors and effectors belonging to the other part of the managed system are assumed to be instrumented in another way. The following sections explain the instrumentation technique used by the framework, and the content of the instrumentation code.

8.4.1 Instrumentation Technique

Aspect-Oriented Programming (AOP) [64] is a programming technique that allows developers to cleanly separate cross-cutting concerns. A concern is a particular functionality provided by a software program. Examples of concerns in the Trading System are Trade Administration or Payment Realisation. Cross-cutting concerns are aspects of a program which affect (cross-cut) other concerns (e.g., logging or authentication concerns touch both Trade Administration and Payment Realisation). Similar to logging, this thesis considers the self-management aspect as a cross-cutting concern [84], and therefore uses AOP to instrument the self-management aspect in the managed system.

The instrumentation locations specified in the generic model (see Section 4.8.1) correspond to the join points in AOP. A join point is a region in the dynamic control flow of a program. There are various join point models depending on the underlying programming language. Examples of join points are: a call to a method, execution of a method, the event of setting a field, or the event of handling an exception. The static projection of a join point onto the program’s source code or compiled code is called a join point shadow. AOP languages implement an aspect by weaving hooks into the join point shadows.

There are different implementations of AOP for the Java language: AspectJ [43], Spring AOP [104], JBoss AOP [98], and Javassist [40]. These AOP implementations differ in the join point models they provide, and whether they require Java source code or not. The current implementation of the proposed framework uses Javassist to automatically instrument that part of a managed system written in the Java language. The reason for choosing Javassist is mainly because it can be used to add self-management aspects to an existing system by directly altering its compiled code (byte-code), without requiring its source code. Note that an existing distributed system usually has been developed by multiple vendors, and its source code is not always available.

For the C++ language, Spinczyk et al. [170] have developed AspectC++, which
The Execution Environment

is an aspect-oriented language extension for C++. The AspectC++ weaver is a source to source weaver that transforms AspectC++ programs into C++ programs. The framework can be easily extended to utilise the AOP implementation for the C++ language (or other object-oriented languages).

8.4.2 Instrumentation Code

The instrumentation code contains code fragments for setting the values of the properties of the specific Sensor class (e.g., amountSensor) and sending these values to the appropriate autonomic manager. The Sensor class has three important properties: observedValue, targetAddress, and timestamp. The first property is the value of the monitored item, the second one contains the address of the autonomic manager waiting for the value of this sensor, and the third one is the actual timestamp of the observation. The observedValue is the result of an invocation to a sensor value provider class that is responsible for gathering the requested information (i.e., the value of the monitored item) about the behaviour of the managed system.

The instrumented code is more or less the same for all specified sensors. For example, Figure 8.4 shows the instrumented code that invokes the amountSensor.

The information concerning the instrumentation location is specified in the OWL domain model. The invocation code is instrumented after the statement containing code which reads the value of the monitored item amount.

```
... read access to the monitored item 'amount' ...
(1) AmountSensor amountSensor = new AmountSensor();
(2) SensorValueProvider svp = AmountSensorValueProvider.getInstance();
(3) Object observedValue = svp.getObservedValue();
(4) amountSensor.setObservedValue(observedValue);
(5) amountSensor.setTimestamp(new Date());
(6) amountSensor.sendSensor();
```

Figure 8.4: The Java code snippet for invoking a sensor implementation (amountSensor implementation).

Line (1) creates an instance of the AmountSensor class generated by the framework. Lines (4) and (5) set the values of the observedValue and timestamp properties. Line (2) retrieves an instance of the AmountSensorValueProvider class that provides the runtime value of the monitored item (line (3)). All sensor value providers (e.g., AmountSensorValueProvider) implement the Java interface, SensorValueProvider, supplied by the framework. Line (6) sends the mentioned information to the given autonomic manager.

The sensor value provider class can be either generated by the framework or supplied by domain experts. If it is generated then the instrumentation process also instruments that class at the given instrumentation location. Otherwise the supplied class is linked into the code of the managed system (i.e. becomes a part of the managed system) in order to be invoked.

An example of the sensor value provider class (AmountSensorValueProvider) that is generated by the framework is shown in Figure 8.5. This class is gener-
ated based on the specifications in the model, and it is instrumented at the given instrumentation location. Line (1) retrieves an instance of the AmountSensorValueProvider class, and line (2) stores the value of the monitored item amount in the sensor value provider class so that it can be used at a later time.

```java
... read access to the monitored item 'amount' ...
(1) SensorValueProvider svp = AmountSensorValueProvider.getInstance();
(2) svp.setObservedValue(amount);
```

Figure 8.5: The Java code snippet showing how a sensor implementation (amountSensor implementation) gathers required information.

Examples of the sensor value provider classes supplied by domain experts are: a class that reads the value of a monitored item from a database or from a log file, a class that checks the status of a connection by sending a Ping request, or a class that collects required information from a sub-system having only binary code.

The starting point for the instrumentation process is the OWL domain model. The process iterates through all specified sensors defined in the OWL model, identifies the instrumentation location in the managed system, and uses Javassist to modify the managed system by instrumenting the appropriate code.

### 8.5 Summary

This chapter explains the execution environment in which self-management of an existing distributed system can be realised. The activation & control engine is the most important component of the execution environment. The different aspects of the activation & control engine such as the communication and synchronisation between autonomic managers, invocation of an inspective plan by an autonomic manager, and the communication between an autonomic manager and the rule engine are explained. The self-management framework utilises the information in the self-management model to enable the code generation tool to automatically generate code for the activation & control engine implementing the autonomic management of a specific managed system.

For the activation & control engine to communicate with a managed system, sensors and effectors need to be instrumented (manually or automatically) in a managed system. The self-management framework utilises the information in the self-management model to enable the instrumentation tool to automatically instrument sensors and effectors in a managed system. The instrumentation process automatically instruments invocations to the sensors in those parts of the managed system that are written in an object-oriented language. The Aspect-Oriented Programming (AOP) technique is used to realise the instrumentation. The domain experts are expected to instrument (manually or using some other instrumentation method) sensors in the remaining parts of the managed system.
Chapter 9

Conclusions and Future Work

This chapter first summarises the work reported in this thesis. Thereafter, the research problem and the central research question outlined in Chapter 1 are revisited. Finally, a number of possible directions for future research work are proposed.

9.1 Thesis Summary

This thesis focuses on management of distributed systems. The growing complexity of these systems makes their management a challenge. A distributed system exposes a considerable number of behaviours, and is composed of a variety of subsystems and components that execute these behaviours. The complexity of both behaviour and structure of these systems is increasing. Management of these systems must deal with both behavioural and structural complexities. To manage these complexities, the general principles of the autonomic computing paradigm are used to let systems manage themselves (self-management). According to this paradigm, a self-managed system has two main building blocks: an autonomic manager and a managed resource. In this thesis, an individual behaviour is the unit of management (i.e., managed resource) for an autonomic manager.

Behaviour is defined as a unit of functionality expected from a system such as handling a money transaction by an Internet banking system or showing articles in response to a search query by a web shopping system. Usually a system shows multiple behaviours. Behaviours of a system can be complex or simple. A complex behaviour consists of a number of less complex or simple behaviours. Showing articles by a web shopping system is an example of a complex behaviour. This behaviour consists of analysing the given search query, retrieving requested data from database, and converting the data into a human readable format. As mentioned before, autonomic managers need to have knowledge about their managed behaviours. The knowledge describing the different behaviours of a system is mainly acquired during the software design and development phase, and specified in use-case notations. This thesis proposes the reuse of use-case specifications,
representing system behaviour, for self-management purposes (Chapter 3).

The activities of autonomic managers are based on the management model for distributed systems presented in this thesis. The management model contains the following functional entities: analyser, diagnoser, planner, and plan translator. It also includes the information flow entities sensor, symptom, hypothesis, plan, and effector. To manage individual system behaviour, an autonomic manager utilises information coming from sensors instrumented in a managed system. The analyser, based on the sensor values, ascertains whether the managed system shows an abnormal behaviour (symptom). The diagnoser determines the possible root-cause of the abnormal behaviour (hypothesis). The planner, based on the diagnosis determined, selects a plan from its plan repository. Thereafter, the plan translator translates the selected plan into executable adaptation instructions (effectors). This process is called autonomic management process. Autonomic managers communicate the result of their autonomic management process with each other to manage the complex behaviours of their managed system (Chapter 4).

For self-managed distributed systems to be able to recognise and solve a large portion of their malfunctionings on their own, they need to know themselves. For this purpose, two models of a managed system are proposed and constructed: behavioural and structural models. The first model describes the way a distributed system provides its functionalities, and the second model describes the internal structure of a distributed system. The elements of the behavioural model are jobs, tasks, states, and events. The underlying idea here is that a system realises a behaviour by means of performing a number of activities (tasks). These activities cause changes in the states of the system. Besides, while performing any activity by the system, events can occur. These events impact the normal performance of other system activities. The elements of the structural model are runnables, connectors, components, classes, and methods. The underlying idea here is that a distributed system consists of multiple processes (runnables), each of which is a running software program. These processes communicate with each other through network software programs (connectors). Each process consists of a number of components. Each component consists of a number of classes. Finally, each class consists of a number of functions (methods) which perform the system activities. Autonomic managers use both models for management purposes (Chapter 5).

All knowledge included in behavioural, structural, and management models is referred to as self-management knowledge. To utilise this knowledge in an automated environment, this knowledge needs to be represented in a knowledge representation language. This thesis discusses a number of important requirements for representing self-management knowledge in distributed environments, and argues that the Semantic Web languages OWL and SWRL together satisfy the requirements. Furthermore, the different parts of self-management knowledge are represented in Semantic Web ontologies (Chapter 6).

A self-managed system consists of an autonomic management part (containing multiple autonomic managers) and a managed system. To enrich an existing distributed system with self-management capabilities, a self-management framework has been designed and implemented. Two case studies have been implemented
9.2 Research Question Revisited

The goal of this thesis is to improve management of the behaviour of existing distributed systems. Manual management of these systems is costly and time-consuming, and suffers from a major drawback: delayed problem determination. In practice, knowledge about a managed system, required for manual management, is most often divided over a number of human administrators with different skills. Hence, manual management, to a large extent, depends on effective communication between members of an administrator team, each with their own technical terminology. The other source of the manual management knowledge is system documentation that is often incomplete and subject to multiple interpretations due to the fact that it is usually written in natural language. Automated management could possibly reduce problem determination time because the knowledge of various human administrators with different skills can be aggregated in a management software layer that can immediately be aware of the occurrence of a problem. In this context, the following central research question was formulated:

**RQ:** How to design autonomic management of behaviour of existing distributed systems?

To answer this question, this thesis investigates the following aspects concerning distributed systems:

- **different types of behaviours to be managed,**
- **the relationship between behaviours,**
- **knowledge required about behaviours and their relationships.**

These aspects have been explored in Chapters 3 and 5. In Chapter 3, behaviours of a distributed system are divided into system, operational, functional, and low level code behaviours. All of these types of behaviours are represented in use-cases notations. Behaviours of the same type are related to each other...
through horizontal use-case references. Behaviours of different types are related
to each other by means of vertical use-case references.

In Chapter 5, generic concepts regarding the structure and behaviour of dis-
tributed systems were abstracted. The relationships between these concepts are
incorporated into the structural and behavioural models. These models contain
the knowledge required for the self-management of distributed systems. Chapter 6
illustrated the representation of these models in the Semantic Web language OWL.

This thesis also investigates the following aspects concerning management of
distributed systems:

- management of an individual behaviour,
- coordination of the management of multiple behaviours,
- knowledge required for the management of behaviours.

These aspects were explored in Chapter 4 in which different components of
the autonomic manager are depicted. Also, the information flow entities (sensor,
symptom, hypothesis, plan, and effector), that are passed from one autonomic
manager’s main entity to the next, were introduced. To realise management of an
individual behaviour of a distributed system, it was specified how an autonomic
manager needs to coordinate the working of all its main entities.

To coordinate management of multiple behaviours, multiple distributed auto-
nomic managers are organised into parent-child hierarchies. After completion of
the autonomic process, the child autonomic manager sends its result to its par-
ent. A parent autonomic manager realises the distributed autonomic process by
integrating the results of its children with its own observations from real-world.

The knowledge required for the management of behaviours are incorporated in
the management model. Chapter 6 illustrated the representation of this model in
the Semantic Web languages OWL and SWRL.

9.3 Discussion

The self-management framework is the important artifact proposed and produced
by this thesis. A number of factors, discussed in the following sections, influences
the practical usage of the framework for the self-management of large existing
distributed systems.

Knowledge Acquisition

Currently, the self-management knowledge cannot be automatically acquired from
the code of the distributed systems. The framework requires specification of the
self-management knowledge by domain experts. The volume of the required knowl-
edge depends on the number of managed behaviours of the system. The concern
is that the high volume of the required knowledge can prevent domain experts
from using the framework. To deal with this concern, a balance should be struck
between coarse-grained behaviours (system level, runnable level, and component level) and fine-grained behaviours (class level) to be managed. Usually, the number of fine-grained behaviours of a distributed system is high and the number of coarse-grained behaviours is low. To cope with the issue, one can suggest to only manage the coarse-grained behaviours. The drawback of managing only coarse-grained behaviours is that autonomic management only performs the coarse-grained diagnosis and adaptation (i.e., the root-cause of a system malfunctioning is related to the whole system, a runnable, or a component). A slightly better suggestion might be to specify only those fine-grained behaviours that are related to the important coarse-grained behaviours. In this way, both the volume of the required knowledge remains acceptable and the fine-grained diagnosis and adaptation can be achieved.

Note that the framework does not require specification of self-management knowledge all at one point in time. Specification of the self-management knowledge is a continuous process. Because of the occurrence of a system malfunctioning, the self-management knowledge regarding a new behaviour can be added into the knowledge-base, or the self-management knowledge regarding an existing behaviour can be refined and updated.

**Synchronisation**

The framework does not provide synchronous communication between autonomic management and the managed system. Accordingly, the managed system does not need to wait for the result of the autonomic process (diagnosis and remedy actions) of autonomic management. The motivation for this design decision is to minimise the influence of autonomic management on the regular working of the managed system. The consequence of this decision is that the first occurrence of a system malfunctioning cannot be prevented. The explanation for this restriction is as follows. The instrumented sensors in the managed system send information regarding the malfunctioning to autonomic management before the malfunctioning occurs. Autonomic management receives the information and takes time to analyse the current situation. As the managed system does not wait for the response of autonomic management, the malfunctioning of the managed system occurs for the first time. After this occurrence, autonomic management performs the diagnosis and prepares effectors to execute remedy actions such that the next occurrence of the same malfunctioning is prevented.

To be able to prevent the first occurrence of a system malfunctioning, the managed system must wait for the autonomic process of autonomic management. This feature does not have a considerable impact on the managed system if the response time of the system is not of significant importance. However, in systems in which a late response is sometimes worse than no response at all, and time is the most precious and critical resource, the synchronous communication between autonomic management and the managed system disturbs the regular working of the managed system (i.e., existing distributed system) substantially.
Performance & Availability of Managed System

Sensors and effectors instrumented in the managed system affect the performance and availability of the managed system. The higher the number of instrumented sensors and effectors, the higher their impact on the performance of the managed system. Moreover, the availability of the managed system can be affected by any critical mistake in the code of the instrumented sensors and effectors provided by domain experts. For example, an instrumented sensor causing a memory problem can lead to a crash of the managed system. This is an inevitable consequence of the autonomic computing control loop and architectural blueprint. Domain experts are expected to pay more attention to the code for the sensors and effectors instrumented in the managed system.

Scalability of Autonomic Management

The scalability of autonomic management can become an issue if, for any reason (such as cost considerations), all autonomic managers are deployed on the same machine. Autonomic management may not be prepared to handle such increased workload. Shared usage of computing and network resources also disturbs the performance of autonomic management. Note that the framework provides the possibility to fully distribute autonomic managers on different machines.

Knowledge Representation

The Semantic Web language OWL is expressive enough to specify the self-management knowledge. However, this is not true for the Semantic Web rule language SWRL. At present, SWRL does not support negated atoms, disjunction, or retraction of axioms. As a result, the construction of more complex logical combinations of atoms in SWRL becomes difficult. Another issue with SWRL is the availability of a stable cross-platform SWRL engine. Currently, other rule engines are used to evaluate the SWRL statements. Hence, the SWRL statements must first be translated into the syntax of the other rule languages, and the inferences of the rule engines must be translated back into the SWRL syntax. Although this back and forth translation is done automatically, it affects the performance of autonomic management. Another solution needs to be considered.

9.4 Future Work

The self-management framework presented in this thesis can be extended in a number of areas briefly discussed in the following sections.

Reasoning

The self-management framework can be extended with other reasoning techniques than rule-based reasoning, such as fuzzy and case-based reasoning. Currently, the framework utilises rule-based reasoning to choose a policy, infer a symptom,
determine a diagnosis, and select a plan. One of the limitations of rule-based reasoning is the difficulty of expressing uncertain knowledge. Another limitation is the inability to deal with a situation where the observation does not exactly match the condition of a rule in the rule base. It can be investigated how to incorporate fuzzy and case-based reasoning into the framework to overcome these limitations.

Planning

The self-management framework can be extended with a proper planning approach. As mentioned in Chapter 4, a plan (either a remedy plan or an inspective plan) is a collection of actions that are bundled by a control construct such as IfThenElse or RepeatUntil. Currently, a domain expert determines which actions by means of which control construct should be bundled. To automatically construct a plan from a given initial state, goal state, and a set of possible actions, different planning approaches such as hierarchical planning [45], multi-agent planning [51], case-based planning [168], heuristic planning [163], etc. can be investigated.

Applicability

The self-management framework can be applied to agent-oriented software systems by integrating the framework in agent platforms. As an example, the framework can be implemented as a middleware service of the AgentScape platform [185]. AgentScape provides a secure and large-scale runtime platform for execution of heterogeneous mobile agents. Agents can use the middleware services of AgentScape to perform their tasks properly. For example, they use the AgentScape services to look up each other, discover other agents’ services, transparently and securely route messages, access web services, and negotiate about the quality of services.

The assumption is that each agent is equipped with its self-management knowledge. When an agent enters the AgentScape platform, the agent provides its self-management knowledge to the self-management middleware service. This service generates autonomic management code for the agent and instruments sensors and effectors in the agent, based on the provided self-management knowledge.
Appendix A

Generic Rules

The autonomic manager’s main entities utilise rules (e.g., SymptomOccurrence-Rules) to determine the occurrence of symptoms, identify diagnoses, select proper plans, and translate plans to executable adaptation instructions. There are two sets of rules: generic rules and domain-specific rules. The domain-specific rules are relevant to the management of a particular distributed system and provided by domain experts. In contrast to the domain-specific rules, the generic rules are domain independent. They are part of the proposed model and are explained in the following sections. The Backus-Naur Form (BNF) [2] is used to denote the generic rules. All nonterminals appear in italic, and start with a capital. All terminals, which are in first-order predicate logic, appear in boldface. Variables are depicted in italics.

The purpose of a rule engine is to continuously apply a set of rules (if-then statements) to a set of data (the knowledge base). A set of rules is stored in the rule memory, and a set of facts is stored in the fact memory. A pattern matcher looks at both sets and generates a set containing rules whose conditions have been satisfied. This set is called the conflict set. A conflict resolver is called to determine which particular rule to fire. Firing rules make some changes in the fact memory (creates new facts or removes old ones). Each time any single rule performs one or more of such changes, the rule engine immediately enters a new cycle. This cycle is called a match-resolve-act\textsuperscript{1} cycle.

A.1 Symptom Occurrence Rules Template

For each managed use-case, domain experts specify various abnormal behaviours as symptoms. They also specify monitored items used within the managed use-case, and sensors that indicate the occurrence of the abnormal behaviours concerning the monitored items. The sensors that indicate the occurrence of a symptom are said to be the relevant sensors of that symptom. Based on the rule template shown

\textsuperscript{1}A simple implementation of the match-resolve-act cycle can be very inefficient. Many rule engines use an efficient implementation based on the Rete algorithm [68].
in Figure A.1, domain experts provide rules that infer the occurrence of a symptom from the information obtained from the relevantSensors of that symptom.

| Symptom-occurrence-rule ::= Expr [∧ Expr]* → arisen(sy,te) |
| Expr ::= Expr1 | Expr2 |
| Expr1 ::= isRelevantSensor(se, sy, true) ∧ observedValue(mi, v1) ∧ (v1 Op1 v2) |
| Expr2 ::= isRelevantSensor(se2, sy, true) ∧ observedValue(mi2, v2) ∧ (v1 Op2 v2) |

where:
- sy belongs to a set of symptoms of AutonomicManager.
- se, se1, se2 belong to a set of sensors of AutonomicManager.
- te is a Ternary value limited to: pos, neg.
- arisen is a relation that is defined as: Symptom × Ternary.
- mi, mi1, mi2 belong to a set of states and events to be monitored.
- v1, v1, v2 belong to one of the ValueDomains such as IntegerType, StringType, etc.
- isRelevantSensor is a relation that is defined as: Sensor × Symptom × BooleanType.
- observedValue is a relation that is defined as: Sensor × (State(Event)) × ValueDomain.
- OpInt is one of operations <, ≤, =, ≥, defined as: IntegerType × IntegerType → BooleanType.
- OpDbl is one of operations <, ≤, =, ≥, defined as: DoubleType × DoubleType → BooleanType.
- OpDate is one of operations <, ≤, =, ≥, defined as: DateType × DateType → BooleanType.
- OpTime is one of operations <, ≤, =, ≥, defined as: TimeType × TimeType → BooleanType.
- OpBool is the operation =, defined as: BooleanType × BooleanType → BooleanType.
- OpChar is the operation =, defined as: CharType × CharType → BooleanType.
- OpString is the operation =, defined as: StringType × StringType → BooleanType.

Figure A.1: The template for the analyser’s symptom occurrence rules.

Each symptom occurrence rule consists of a logical implication (see [153] for an introduction of formal logic). The antecedent is composed of a number of logical conjunctions of expressions, and the consequent is the Symptom’s arisen attribute that is used as a logical relation. The entities used in the expressions are Symptom, Sensor, MonitoredItem, and ValueDomain.

Initially, the analyser assumes that it is unknown whether a symptom has occurred. To conclude the occurrence or absence of a symptom sy, the analyser first checks whether the sensor (se) belongs to the relevantSensors of sy. Then the observed value (v1) of the MonitoredItem of the sensor is compared with either a pre-defined constant value vse (Expr1 in the figure) or the observed value of the monitored item of another sensor (Expr2 in the figure). In the first case, the value of a state (variable) or an event is compared with the expected value defined by domain experts. In the second case, the dependency of two or more states (or events) is checked by comparing the observed values of different sensors. The values v1, v1, v2 belong to one of the ValueDomains (see Section 4.8.1) depending on the type of the monitored item (i.e., the specific state or event type defined in the system model). The comparison operation between v1 and v2 takes place if v1 and v2 have the same ValueDomain type. The usual comparison operators (<, ≤, =, ≥) can be applied to variables of all ValueDomain types except for BooleanType, CharType, and StringType. Only the equality operator (=) is allowed to be applied to variables of these three types.
A.2 Hypothesis Selection Rules

The generic rules in Figure A.2 illustrate the logical expressions used by the hypothesis selection rules. These rules that can be used for all distributed systems are denoted by $GRule_i$. The $userDefSelCriteria$ is an attribute of Hypothesis, and represents the domain-specific criteria. The value of this attribute is initially set to $true$. This allows the execution of the generic rules without incorporating the domain-specific rules. The assumption is that domain experts provide a set of rules that determine the value ($true$ or $false$) of $userDefSelCriteria$. The value of the domain-specific criteria is combined with the relations used in the generic rules in order to select a hypothesis for validation purposes. The diagnostic engine executes the domain-specific rules before the generic rules. Both rules are executed for all available hypotheses.

\[
Hypothesis-selection-rule := GRule_1 | GRule_2 | GRule_3 | GRule_4 | GRule_5 | GRule_6 | GRule_7
\]

\[
GRule_1 := \text{validated}(\text{hy}, \text{neg}) \rightarrow \text{tried}(\text{hy}, \text{true})
\]

\[
GRule_2 := \text{assessed}(\text{hy}, \text{true}) \rightarrow \text{tried}(\text{hy}, \text{true})
\]

\[
GRule_3 := \text{tried}(\text{hy}, \text{true}) \rightarrow \text{toBeFocussed}(\text{hy}, \text{false})
\]

\[
GRule_4 := \text{numUnknownSymptoms}(\text{hy}, v_1) \land \text{allowedUnknownSymptoms}(\text{hy}, v_2) \land (v_1 \leq v_2) \rightarrow \text{toBeFocussed}(\text{hy}, \text{true})
\]

\[
GRule_5 := \text{toBeFocussed}(\text{hy}, \text{false}) \rightarrow \text{focussed}(\text{hy}, \text{false})
\]

\[
GRule_6 := \text{toBeFocussed}(\text{hy}, \text{true}) \land \text{userDefSelCriteria}(\text{hy}, \text{true}) \rightarrow \text{focussed}(\text{hy}, \text{true})
\]

\[
GRule_7 := \text{toBeFocussed}(\text{hy}, \text{true}) \land \text{userDefSelCriteria}(\text{hy}, \text{false}) \rightarrow \text{focussed}(\text{hy}, \text{false})
\]

where:
- \text{validated} is a relation that is defined as: Hypothesis $\times$ Ternary.
- \text{assessed} is a relation that is defined as: Hypothesis $\times$ BooleanType.
- \text{tried} is a relation that is defined as: Hypothesis $\times$ BooleanType.
- \text{toBeFocussed} is a relation that is defined as: Hypothesis $\times$ BooleanType.
- \text{numUnknownSymptoms} is a relation that is defined as: Hypothesis $\times$ IntegerType.
- \text{allowedUnknownSymptoms} is a relation that is defined as: Hypothesis $\times$ IntegerType.
- \text{subHypothesis} is a relation that is defined as: Hypothesis $\times$ Hypothesis $\times$ BooleanType.
- \text{userDefSelCriteria} is a relation that is defined as: Hypothesis $\times$ BooleanType.
- \text{focussed} is a relation that is defined as: Hypothesis $\times$ BooleanType.

Figure A.2: The generic rules for the diagnoser’s hypothesis selection.

The goal is to select one or more hypotheses from the hypothesis set of the current autonomic manager, and mark them as focussed in order to be further validated. Initially, all hypotheses are marked as not focussed. The set contains both hypotheses that have already been tried and hypotheses that have never been examined. A hypothesis is considered as tried if it has already been validated and/or assessed by the other rules of the diagnoser.

Figure 4.6 shows that there are three closed loops from the SelectingHypotheses state: (1) a loop that goes through the ValidatingHypotheses state and comes back to the SelectingHypotheses state if the selected hypothesis is invalid, (2) a loop that goes through the ValidatingHypotheses and EvaluatingHypotheses states, and comes back to the SelectingHypotheses state if the validated hypothesis is assessed, (3) a loop that goes through the ValidatingHypotheses, ExecutingInspectivePlan, ExecutingStrategicRules and MappingChildResultToSymptoms states, and comes back to the SelectingHypotheses state if the inspective plan execution clarifies the occurrence or absence of a symptom. Therefore, the HypothesisSelectionRules should take into account that the values of the hypothesis’s attributes (i.e., the
value of focussed, validated, and assessed) could change during previous activations of the mentioned loops.

The generic rules in Figure A.2 show that a hypothesis (hy) is considered as a candidate to be focussed if (1) the number of unknown symptoms of hy is not greater than a pre-defined threshold, (2) hy has not been already validated, and (3) hy has not already been evaluated. The selection result is subsequently influenced by the domain-specific selection criteria. The numUnknownSymptoms is an attribute of the hypothesis hy, and indicates the number of unknown symptoms of hy. It is initially set to the number of symptoms of hy. The diagnoser is responsible for decreasing its value each time the occurrence of a symptom, belonging to the relevantSymptoms of hy, becomes known. The allowedUnknownSymptoms is also an attribute of the hypothesis hy, and indicates the maximum number of symptoms of hy allowed to be unknown. Its value is initially determined by a policy rule for all hypotheses. Domain experts can override that value for a specific hypothesis.

A.3 Hypothesis Validation Rules

Initially, the diagnoser sets the value of the validated attribute of all hypotheses to unknown. The hypothesis validation rules, illustrated in Figure A.3, set the value of validated of the focussed hypotheses either to pos or to neg. It is also possible that the rules leave the value of validated unchanged and just activate the inspective plans. The following explains the generic rules.

\[
\begin{align*}
\text{Hypothesis-validation-rule} & ::= GRule_1 | GRule_2 | GRule_3 | GRule_4 | GRule_5 | GRule_6 | GRule_7 \\
GRule_1 & ::= (\exists y_i \ isRelevantSymptom(y_i, hy, true) \rightarrow \mathit{arisen}(y_i, te)) \\
& \land \ \mathit{focussed}(hy, true) \rightarrow \mathit{toBeValidated}(hy, true) \\
GRule_2 & ::= (\exists y_i \ isRelevantSymptom(y_i, hy, true) \rightarrow \mathit{arisen}(y_i, unknown)) \\
& \land \ \mathit{focussed}(hy, true) \land \mathit{existInspectivePlan}(y_i, false) \rightarrow \mathit{validated}(hy, neg) \\
GRule_3 & ::= (\exists y_i \ isRelevantSymptom(y_i, hy, true) \rightarrow \mathit{arisen}(y_i, unknown)) \\
& \land \ \mathit{focussed}(hy, true) \land \mathit{isInspectivePlan}(ip, sy, true) \\
& \rightarrow \mathit{toBeInspected}(ip, sy, hy, true) \\
GRule_4 & ::= \mathit{toBeInspected}(ip, sy, hy, true) \land \mathit{inspectivePlanExecuted}(ip, false) \\
& \rightarrow \mathit{activated}(ip, true) \\
GRule_5 & ::= \mathit{toBeInspected}(ip, sy, hy, true) \land \mathit{inspectivePlanExecuted}(ip, true) \\
& \rightarrow \mathit{validated}(hy, neg) \\
GRule_6 & ::= \mathit{toBeValidated}(hy, true) \land \mathit{userDefValCriteria}(hy, true) \rightarrow \mathit{validated}(hy, pos) \\
GRule_7 & ::= \mathit{toBeValidated}(hy, true) \land \mathit{userDefValCriteria}(hy, false) \rightarrow \mathit{validated}(hy, neg) \\
\end{align*}
\]

where:
- te is a Ternary value limited to: pos, neg
- isRelevantSymptom is a relation that is defined as: Symptom \times Hypothesis \times BooleanType.
- toBeValidated is a relation that is defined as: Hypothesis \times BooleanType.
- validated is a relation that is defined as: Hypothesis \times Ternary.
- existInspectivePlan is a relation that is defined as: Symptom \times BooleanType.
- isInspectivePlan is a relation that is defined as: InspectivePlan \times Symptom \times BooleanType.
- toBeInspected is a relation, defined as: InspectivePlan \times Hypothesis \times BooleanType.
- inspectivePlanExecuted is a relation that is defined as: InspectivePlan \times BooleanType.
- activated is a relation that is defined as: InspectivePlan \times BooleanType.
- userDefValCriteria is a relation that is defined as: Hypothesis \times BooleanType.

Figure A.3: The generic rules for the diagnoser’s hypothesis validation.

The GRule1 expresses that the focussed hypothesis hy is the candidate for validation if the occurrence (or absence) of all symptoms sy, belonging to the
relevantSymptoms of hy are known (i.e., the value of the arisen attribute of none of syi is unknown). The GRule2 states that if occurrence (or absence) of at least one of the symptoms is unknown, and there is no inspective plan associated with that symptom then validated of the hypothesis hy is set to neg. However, if there is an inspective plan (GRule3) and it has not been executed during the current diagnostic process then the value of validated of the hy remains unknown and the diagnostic engine activates the inspective plan (GRule4). Finally, the GRule5 expresses that the value of validated of the hy is set to neg if the inspective plan has already been executed but it has not been succeeded to make the occurrence (or absence) of the syi known.

The userDefValCriteria is an attribute of Hypothesis, and its value represents the result of the execution of domain specific validation criteria provided by domain experts. The value of this attribute is initially set to true. The diagnostic engine first executes the set of domain-specific rules that determine the value (true or false) of userValSelCriteria. Thereafter, the generic rules are executed to determine whether the value of validated of the hy is set to pos or not.

### A.4 Hypothesis Evaluation Rules

The goal of the hypothesis evaluation rules is to accept or reject a validated hypothesis hy based on domain-specific evaluation criteria. The attribute userDefAEvalCriteria of a hypothesis hy represents the result of the execution of a set of rules expressing the criteria for accepting the validated hypothesis. The attribute userDefREvalCriteria represents the result of the execution of rules expressing the criteria for rejecting the validated hypothesis. After the execution of these rules, the generic rules shown in Figure A.4 are executed.

- **rejected** is a relation that is defined as: Hypothesis × BooleanType.
- **accepted** is a relation that is defined as: Hypothesis × BooleanType.
- **assessed** is a relation that is defined as: Hypothesis × BooleanType.
- **userDefAEvalCriteria** is a relation that is defined as: Hypothesis × BooleanType.
- **userDefREvalCriteria** is a relation that is defined as: Hypothesis × BooleanType.

![Hypothesis-evaluation-rule](image)

**Figure A.4:** The generic rules for the diagnoser’s hypothesis evaluation.

The GRule1 and GRule2 state that the hy is immediately rejected if the value of validated of the hy is either unknown or neg. The GRule3 through GRule6
set the value of the accepted or rejected attributes of the hy to true or false, based on the values of the domain-specific criteria.

The GRule7 and GRule8 express that the value of assessed of the hy is set to true if the hy is either accepted or rejected. The hypothesis selection rules use the value of this attribute of the hypothesis hy to ignore it from being selected.

A.5 Diagnosis Determination Rules

The last rules to be executed in a diagnostic process are diagnosis determination rules, illustrated in Figure A.5. These rules are responsible for providing the result of the diagnostic process, namely one or more diagnoses. If a hypothesis hy has already been rejected then it is deleted from the set of possible diagnoses (GRule1). The hy is immediately marked as the diagnosis if it has been accepted by the hypothesis evaluation rules, and if it is the only member of the set of accepted hypotheses (GRule2).

\[
\text{Diagnosis-determination-rule ::= GRule}_1 \mid GRule_2 \mid GRule_3 \mid GRule_4 \mid GRule_5 \mid GRule_6
\]

\[
\text{GRule}_1 ::= \text{rejected}(hy, true) \rightarrow \text{determined}(hy, false)
\]

\[
\text{GRule}_2 ::= \text{accepted}(hy, true) \rightarrow \text{determined}(hy, false)
\]

\[
\text{GRule}_3 ::= \text{accepted}(hy_1, true) \land \text{accepted}(hy_2, true) \land \text{weight}(hy_1, v_1) \land \text{weight}(hy_2, v_2)
\]

\[
\land (v_1 \ge v_2) \rightarrow \text{toBeDetermined}(hy_1, true)
\]

\[
\text{GRule}_4 ::= \text{toBeDetermined}(hy, true) \land \text{subHypothesis}(hy_1, hy_2, true) \rightarrow \text{determined}(hy_2, false)
\]

\[
\text{GRule}_5 ::= \text{toBeDetermined}(hy, true) \land \text{userDefDetCriteria}(hy, true)
\]

\[
\rightarrow \text{determined}(hy, true)
\]

\[
\text{GRule}_6 ::= \text{toBeDetermined}(hy, false) \land \text{userDefDetCriteria}(hy, false)
\]

\[
\rightarrow \text{determined}(hy, false)
\]

where:
- toBeDetermined is a relation that is defined as: Hypothesis $\times$ BooleanType.
- weight is a relation that is defined as: Hypothesis $\times$ IntegerType.
- userDefDetCriteria is a relation that is defined as: Hypothesis $\times$ BooleanType.
- determined is a relation that is defined as: Hypothesis $\times$ BooleanType.

Figure A.5: The generic rules for the diagnoser’s diagnosis determination.

In case the accepted hypothesis set contains more than one hypothesis, then the pre-defined weights of the hypotheses, given by domain experts, are compared and the one with the highest weight is chosen as the diagnosis (GRule3). In addition, the determination rules prefer a parent hypothesis hy2 (more generic root-cause) to a child hypothesis hy1 (more specific root-cause) (GRule4). Finally, the value of the userDefDetCriteria attribute, determined by a set of domain-specific diagnosis determination rules, affect the result.

A.6 Plan Selection Rules

Figure A.6 shows the rules that select the remedy plans based on the appropriate diagnoses. The GRule1 states that plan pl is chosen as the candidate to be translated (and thereafter to be performed) if the pl has been specified by domain experts as the suitable remedy plan for the current system malfunctioning whose
root-cause is the diagnosis \( hy \). The \( GRule_2 \) checks to see whether the candidate plan \( pl \) is the only available remedy plan. If there is more than one remedy plan, the pre-defined weights of the plans are compared and the one with the highest weight is marked as a suitable remedy plan (\( GRule_3 \)).

\[
\text{Plan-selection-rule} ::= GRule_1 | GRule_2 | GRule_3 | GRule_4 | GRule_5 \\
GRule_1 ::= \text{determined}(hy, true) \land \text{isRelevantDiagnosis}(hy, pl, true) \\
GRule_2 ::= \text{toBeTranslated}(pl, true) \land (\forall pl_i, \text{toBeTranslated}(pl_i, true) \rightarrow (pl_i = pl)) \\
GRule_3 ::= \text{toBeTranslated}(pl_1, true) \land \text{toBeTranslated}(pl_2, true) \land \text{weight}(pl_1, v_1) \land \text{weight}(pl_2, v_2) \land (v_1 \geq v_2) \rightarrow \text{toBeSelected}(pl_2, false) \\
GRule_4 ::= \text{toBeSelected}(pl, true) \land \text{userDefPlnCriteria}(pl, true) \rightarrow \text{selected}(pl, true) \\
GRule_5 ::= \text{toBeSelected}(pl, true) \land \text{userDefPlnCriteria}(pl, false) \rightarrow \text{selected}(pl, false)
\]

where:
- \( \text{isRelevantDiagnosis} \) is a relation that is defined as: \( \text{Hypothesis} \times \text{Plan} \times \text{BooleanType} \).
- \( \text{toBeSelected} \) is a relation that is defined as: \( \text{Plan} \times \text{BooleanType} \).
- \( \text{toBeTranslated} \) is a relation that is defined as: \( \text{Plan} \times \text{BooleanType} \).
- \( \text{weight} \) is a relation that is defined as: \( \text{Plan} \times \text{IntegerType} \).
- \( \text{userDefPlnCriteria} \) is a relation that is defined as: \( \text{Plan} \times \text{BooleanType} \).
- \( \text{selected} \) is a relation that is defined as: \( \text{Plan} \times \text{BooleanType} \).

Figure A.6: The generic rules for the planner’s plan selection.

Note that it is possible that a domain expert defines more than one remedy plan based on one diagnosis. Depending on the situation expressed as the domain-specific rules, one of the remedy plans is selected.

The value of \( \text{userDefPlnCriteria} \), an attribute of \( \text{Plan} \), is the result of the execution of the domain-specific rules expressing the plan selection criteria out of the set of remedy plans. This value finally determines which remedy plans should be translated.

### A.7 Plan Translation Rules

The goal of \( \text{PlanTranslationRules} \), shown in Figure A.7, is to map a selected remedy plan to an effector. The generic rule \( GRule_1 \) checks to see whether the basic condition for choosing the appropriate effector is satisfied. The effector \( ef \) is chosen if the planning process has indicated \( pl \) as the remedy plan for compensating the current system malfunctioning, and if domain experts defined a relation between \( pl \) and \( ef \).

\[
\text{Plan-translation-rule} ::= GRule_1 | GRule_2 | GRule_3 \\
GRule_1 ::= \text{selected}(pl, true) \land \text{isRelevantPlan}(pl, ef, true) \rightarrow \text{toBeActive}(ef, true) \\
GRule_2 ::= \text{toBeActive}(ef, true) \land \text{userDefEffCriteria}(ef, true) \rightarrow \text{isActive}(ef, true) \\
GRule_3 ::= \text{toBeActive}(ef, true) \land \text{userDefEffCriteria}(ef, false) \rightarrow \text{isActive}(ef, false)
\]

where:
- \( \text{isRelevantPlan} \) is a relation that is defined as: \( \text{Plan} \times \text{Effector} \times \text{BooleanType} \).
- \( \text{toBeActive} \) is a relation that is defined as: \( \text{Effector} \times \text{BooleanType} \).
- \( \text{isActive} \) is a relation that is defined as: \( \text{Effector} \times \text{BooleanType} \).
- \( \text{userDefEffCriteria} \) is a relation that is defined as: \( \text{Effector} \times \text{BooleanType} \).

Figure A.7: The generic rules for the plan translator’s plan translation.
Similar to other generic rules, the domain-specific criteria for activating an effector $ef$ are represented as the `userDefEffCriteria` attribute of `Effector`. At last, its value determines whether the `isActive` attribute of $ef$ is set to `true` or `false`. 
Appendix B

Autonomic Management Code

B.1 Performing Autonomic Process

The following Java code shows what an autonomic manager performs when it is started, how it starts its children, how it handles the sensors received, how it performs its autonomic process, and how it handles the result of its child. This generic code is used by all specific autonomic managers.

```java
public abstract class AutonomicManager extends Observable implements Observer {
    ...}

protected abstract void setChildren();

/** An autonomic manager starts all its children. */
protected void startChildren() {
    String AMName = this.getClass().getSimpleName();
    for (Iterator<AutonomicManager> iterator = subAutonomicManagers.iterator(); iterator.hasNext();) {
        AutonomicManager child = iterator.next();
        String childName = child.getClass().getSimpleName();
        logger.info("Creating and starting childAM "+childName+" of "+AMName+". ");
        child.start(this);
    }
}

/** The starting point for an autonomic manager. */
public void start(AutonomicManager parentAM) {
    String AMName = this.getClass().getSimpleName();
    String jobName = job.getClass().getSimpleName();
    this.init();
    parentAutonomicManagers.add(parentAM);
    try {
        addObserver(parentAM);
        addObserver(parentAM);
    } catch (Exception e) {
        logger.info("This is probably the MasterAM which has no parent.");
    }
    setChildren();
    startChildren();
    logger.info("Starting Job "+jobName+" for AM "+AMName+"...");
    job.start(this);
    analyser.start();
    diagnoser.start();
}
```
/* The autonomic process is performed and the result is delivered to the parent. */
private void performAutonomicProcess() {
    String AMName = this.getClass().getSimpleName();
    logger.info("Starting autonomic process for AM "+ AMName + ". . .");
    analyser.performAnalysis(sensors, symptoms);
    logger.info("Performing Analysis Rules for AM "+ AMName + ". . .");
    diagnoser.performDiagnosis(hypotheses, symptoms);
    if (countObservers() >= 1) {
        setChanged();
        notifyObservers(autonomicProcessResult);
    }
}

/* A parent autonomic manager handles the result of its child. */
private void handleChildResult(AutonomicProcessResult childResult) {
    String childResultName = childResult.getClass().getSimpleName();
    logger.info("childResult " + childResultName + " received for: "+ AMName + ". . .");
    for (Iterator<AutonomicManager> iterator = subAutonomicManagers.iterator(); iterator.hasNext();) {
        AutonomicManager childAM = iterator.next();
        AutonomicProcessResult childAPR = childAM.autonomicProcessResult;
        if (childAPR.getClass().getSimpleName().equalsIgnoreCase(childResultName)) {
            logger.info("Performing Mapping Rules for AM "+ childAM.getClass().getSimpleName() + ". . .");
            childAM.diagnoser.performMappingRules(hypotheses, symptoms);
            if (jobEndSensorReceived) {
                logger.info("Perform autonomic process again because of new child results.");
                performAutonomicProcess();
            }
        }
    }
    childResult.setHandledByParent(true);
}

/* An autonomic manager handles the received sensor values. */
public void update(Observable observable, Object signal) {
    if (signal == null) {
        logger.error("Signal should not be Null.");
        return;
    }
    if (signal instanceof JobStartSensor) {
        handleJobStart((JobStartSensor)signal);
    } else if (signal instanceof AutonomicProcessResult) {
        handleChildResult((AutonomicProcessResult)signal);
    } else if (signal instanceof JobEndSensor) {
        handleJobEnd((JobEndSensor)signal);
    } else if (signal instanceof Sensor) {
        handleContentSensor((Sensor)signal);
    } else {
        logger.error("AM cannot handle this signal: " + signal.getClass().getSimpleName());
    }
}

B.2 Instantiating Autonomic Manager

The following Java code shows how a specific autonomic manager provides the required information used by the abstract autonomic manager.
B.3 Performing Diagnostic Process

The following Java code shows the implementation of the diagnostic loop where the hypothesis selection, validation, evaluation, and the diagnosis determination rules are executed. This generic code is used by all specific diagnosers.

```java
public abstract class Diagnoser {
    public abstract void performDiagnosis(ArrayList<Hypothesis> hypotheses, ArrayList<Symptom> symptoms) {
        OWLModelHelper.activateHypotheses(hypotheses);
        boolean anyHypSelected = false;
        do {
            logger.info("Performing Hypothesis Selection Rules ...");
            anyHypSelected = OWLModelHelper.inferHypSelRules(hypothesisSelectionRules);
            OWLModelHelper.clearHypotheses(hypotheses, OWLModelHelper.TO_BE_FOCUSSED);
            OWLModelHelper.clearHypotheses(hypotheses, OWLModelHelper.TRIED);
            boolean anyHypValidated = false;
            if (anyHypSelected) {
                logger.info("Performing Hypothesis Validation Rules ...");
                anyHypValidated = OWLModelHelper.inferHypValRules(hypothesisValidationRules, symptoms);
                OWLModelHelper.clearHypotheses(hypotheses, OWLModelHelper.TO_BE_VALIDATED);
            }
            if (anyHypValidated) {
                logger.info("Performing Hypothesis Evaluation Rules ...");
                OWLModelHelper.inferHypEvalRules(hypothesisEvaluationRules);
            }
        } while (anyHypSelected);
        logger.info("Performing Diagnosis Determination Rules ...");
        OWLModelHelper.inferHypDetRules(diagnosisDeterminationRules);
        OWLModelHelper.deactivateHypotheses(hypotheses);
    }
}
```
The following Java code shows how a specific diagnoser provides the various rule names to the abstract diagnoser.

```java
public class SpecificDiagnoser extends Diagnoser {
    private static SpecificDiagnoser specificDiagnoser = null;
    private SpecificDiagnoser() {
    }
    public static SpecificDiagnoser getInstance() {
        if (specificDiagnoser == null) {
            specificDiagnoser = new SpecificDiagnoser();
        }
        return specificDiagnoser;
    }

    /* The various rule names are delivered to the rule engine. */
    public void init() {
        super.init();
        hypothesisSelectionRules.add("specificHypSelRule-1");
        hypothesisValidationRules.add("specificHypValRule-1");
        hypothesisEvaluationRules.add("specificHypEvalRule-1");
        diagnosisDeterminationRules.add("specificHypDetRule-1");
        childResultToSymptomRules.add("specific2AuthPrepMapRule-1");
    }
}
```

### B.4 Handling Sensor Values

The following Java code shows how a job delegates the received sensor values to its associated autonomic manager. This generic code is used by all specific jobs.

```java
public abstract class Job extends Observable implements Observer {
    ...
    public abstract void start(AutonomicManager associatedAM);

    /* A job delegates the received sensor values to its associated autonomic manager. */
    public void update(Observable observable, Object sensor) {
        setChanged();
        notifyObservers(sensor);
    }
}
```

The following Java code shows how a specific job registers its associated autonomic manager and starts the different sensors to listen to their incoming values from the managed system. The abstract job class handles the delegation of the sensor values to the proper autonomic manager.

```java
public class SpecificJob extends Job {
    private static SpecificJob specificJob = null;
    private SpecificJob() {
    }
    public static SpecificJob getInstance() {
        if (specificJob == null) {
            specificJob = new SpecificJob();
        }
        return specificJob;
    }

    /* This starts listening to incoming sensor values. */
    public void start(AutonomicManager associatedAM) {
    }
```
addObserver(associatedAM);
SpecificJobStart SpecificJobStart = new SpecificJobStart();
SpecificJobStart.addObserver(this);
SpecificJobStart.listen();
ContentSensor contentSensor = new ContentSensor();
contentSensor.addObserver(this);
contentSensor.listen();
SpecificJobEnd SpecificJobEnd = new SpecificJobEnd();
SpecificJobEnd.addObserver(this);
SpecificJobEnd.listen();
}
}

B.5 Execution of Rules by Rule Engine

The following Java code shows how an autonomic manager connects to the rule engine, delivers the generic and domain specific rules to the rule engine, and obtains the inferred facts. In addition, it is shown how the autonomic manager starts an inspective plan.

```java
public class OWLModelHelper {
    /* Load proper rules. */
    public static boolean inferHypValRules(ArrayList<String> hypValRuleNames, ArrayList<Symptom> symptoms) {
        String[] ruleNames = new String[hypValRuleNames.size()];
        enableRules(GENERIC_HYP_VAL_RULES_1);
        String hypValNames = new String[hypValRuleNames.size()];
        enableRules(hypValRuleNames.toArray(ruleNames));
        Set<OWLAxiom> inferredProps = inferRules();
        disableRules(GENERIC_HYP_VAL_RULES_1);
        determineInspection(inferredProps, symptoms);
        inferredProps = inferRules();
        disableRules(GENERIC_HYP_VAL_RULES_2);
        return anyHypValidated(inferredProps);
    }

    /* Starting an inspective plan. */
    private static void startInspection(Symptom symptom) {
        if (symptom.isInspectionPerformed()) {
            return;
        }
        else {
            symptom.setInspectionPerformed(true);
            String symptomName = symptom.getInspectivePlan().getModel().getLabel()
            logger.info("@\(\)Inspective Plan " + plan.getClass().getName() + ", associated with the symptom " + symptomName + ", is going to be executed..." + plan.execute();
    }

    /* Perform rule inference. */
    private static Set<OWLAxiom> inferRules() {
        Set<OWLAxiom> inferredProps = new HashSet<OWLAxiom>();
        try {
            bridge.importSWRLRulesAndOWLKnowledge();
            bridge.run();
            logger.info("NumberOfInferredProperties: " + bridge.getNumberofInferredAxioms());
            inferredProps = bridge.getInferredAxioms();
            for (Iterator<OWLAxiom> iterator = inferredProps.iterator(); iterator.hasNext();)
```
logger.info(inferredProp.toString());
}
bridge.writeInferredKnowledge2OWL();
bridge.reset();
}
catch (Throwable t) {
    logger.error("SWRL RuleEngine exception occurred during infering rules.", t);
}

Een informatiesysteem wordt gedefinieerd als een verzameling van hardware, software, netwerk, mensen en procedures die samen gegevens verzamelen en dit vervolgens naar informatie omzetten. Informatiesystemen vertonen een gedrag (leveren een functionaliteit) en hebben een structuur. In dit proefschrift ligt de focus op het beheer van bestaande gedistribueerde informatiesystemen. Bij het beheer van deze systemen moet rekening gehouden worden met zowel gedraggerelateerde als structuurgerelateerde complexiteiten. Om deze complexiteiten aan te kunnen worden in dit proefschrift de generieke principes van het autonomic computing paradigma gebruikt: laat systemen zichzelf beheren (self-management). Volgens dit paradigm heeft een zichzelf beherend systeem twee bouwstenen: een autonomic manager en een managed resource. In dit proefschrift wordt een individueel gedrag van het systeem beschouwd als de basiseenheid (managed resource) die door een autonomic manager wordt beheerd.

Aangezien een gedrag de basiseenheid van autonomic management in de aanpak in dit proefschrift is, is het van belang om te weten wat er met een gedrag bedoeld wordt en hoe een autonomic manager kennis over het te beheren gedrag kan verkrijgen. In essentie is een gedrag een functionaliteit die van een systeem wordt verwacht, zoals het afhandelen van een geldtransactie door een internetbankiersysteem of het tonen van artikelen in antwoord op een zoekopdracht door een webwinkelsysteem. Normaliter vertoont een systeem meervoudige gedragingen. Gedragingen van een systeem kunnen complex of simpel zijn. Een complex gedrag bestaat uit een aantal minder complexe of atomaire gedragingen. Het tonen van artikelen door een webwinkelsysteem is een voorbeeld van een complex gedrag.
gedrag. Dit gedrag bestaat uit het analyseren van de gegeven zoekopdracht, het ophalen van de desbetreffende gegevens vanuit de gegevensbank en het opmaken (leesbaar maken) van de gegevens voor gebruikers. Zoals eerder gezegd moeten autonomic managers kennis hebben over de te beheren gedragingen. De kennis die verschillende gedragingen van een systeem beschrijft wordt vaak verworven tijdens de software ontwerp- en ontwikkel fase en wordt vaak in use-case notaties gespecificeerd. In dit proefschrift wordt voorgesteld om de use-case specificaties, die systeemm gedragingen representeren, te hergebruiken voor bewerkstellig van self-management (Hoofdstuk 3).

De activiteiten van autonomic managers zijn gebaseerd op het management model voor gedistribueerde systemen. Dit model is uitvoerig in dit proefschrift beschreven. Het bevat de volgende functionele entiteiten: analyser, diagnoser, planner en plan translator. Het bevat ook de informatiestroomentiteiten: sensor, symptom, hypothesis, plan en effector. Om een individueel gedrag van een systeem te kunnen beheren maakt de autonomic manager gebruik van de informatie die door sensoren wordt geleverd. Deze sensoren zijn al in het te beheren systeem geïnstrumenteerd. De analyser constateert aan de hand van de sensorwaarden of een afwijkend gedrag (symptom) heeft plaatsgevonden. De diagnoser stelt de eventuele oorzaak (hypothesis) van het afwijkende gedrag vast. De planner selecteert aan de hand van de vastgestelde diagnose een plan vanuit zijn plannen opslagplaats. Vervolgens vertaalt de plan translator het geselecteerde plan naar uitvoerbare adaptatie-instructies (effectors). Dit proces wordt het autonomic management process genoemd. Autonomic managers communiceren het resultaat van hun autonomic management process aan elkaar om de complexe gedragingen van een systeem te beheren (Hoofdstuk 4).

Om zelf-beherend systemen een groot deel van hun eigen gebreken te laten lokaliseren en oplossen is het noodzakelijk dat deze systemen zichzelf kennen. Hier voor zijn twee modellen van een te beheren systeem voorgesteld en geconstrueerd: behavioural en structural modellen. Het eerste model beschrijft hoe een gedistribueerd systeem zijn functionaliteit aanbiedt en het tweede model beschrijft de interne structuur van een gedistribueerd systeem. De elementen van het behavioural model zijn job, task, state en event. Het achterliggende idee hierbij is dat een systeem een gedrag (job) door middel van het uitvoeren van een aantal taken (tasks) realiseert. Deze taken veranderen de verschillende status (state) van het systeem. Tijdens het uitvoeren van een taak kunnen gebeurtenissen (events) plaatsvinden waardoor de normale gang van zaken wordt verstoord. De elementen van het structural model zijn runnable, connector, component, class en method. Het achterliggende idee hierbij is dat een gedistribueerd systeem bestaat uit meerdere processen (runnables), welke een draaiend softwareprogramma is. Deze processen communiceren met elkaar middels netwerk-software programma’s (connectors). Ieder proces bestaat uit een aantal componenten (components). Iedere component bestaat uit een aantal klassen (classes). Ten slotte bestaat iedere klasse uit een aantal functies (methods) die de echte systeemtaken uitvoeren. Autonomic managers gebruiken beide modellen voor hun managementdoelen (Hoofdstuk 5).

Alle kennis die in de behavioural, structural en management modellen is opge

Zoals eerder gezegd bestaat een zelf-beherend systeem uit een aantal autonomic managers (die samen een autonomic management laag vormen) en het te beheren systeem. Om een bestaand gedistribueerd systeem te verrijken met self-management capabiliteiten moet er een autonomic management laag beschikbaar zijn. Voor dit doel is er in dit proefschrift een self-management raamwerk ontworpen en geïmplementeerd. Om de toepasbaarheid van het self-management raamwerk op de management van in de praktijk voorkomende problemen te demonstreren zijn er twee praktijkstudies geïmplementeerd. In de eerste praktijkstudie wordt getoond hoe de autonomic management laag de diagnose van twee gelijktijdig optredende fouten vaststelt. In de tweede praktijkstudie wordt gedemonstreerd hoe autonomic managers in een ouder-kind hiërarchie met elkaar communiceren en coöpereren om een diagnose op meerdere niveaus (multi-level diagnose) mogelijk te maken (Hoofdstuk 7).

Het raamwerk genereert broncode voor de autonomic management laag en voor het instrumenteren van sensoren in het te beheren systeem. Het raamwerk maakt gebruik van de generieke self-management ontologieën en de specifieke instanties van die ontologieën die te beheren systeem door domeinexperts zijn aangemaakt (Hoofdstuk 8).
Bibliography


web service orchestration by means of diagnosis. In European Workshop on 
Software Architecture (EWSA '06), pages 2–16, 2006.

dependence graph and its application to fault diagnosis. In Proceedings of the 
international symposium on Software testing and analysis (ISSTA '08), pages 189–200, New York, NY, USA, 2008. ACM.

ponent model for self-adaptation. In Third IEEE International Conference 
on Self-Adaptive and Self-Organizing Systems (SASO '09), pages 206–215, 
September 2009.

[14] L. Baresi and S. Guinea. Towards dynamic monitoring of WS-BPEL pro-
ces. In Proceedings of the 3rd Int. Conference on Service Oriented Com-
puting (ICSOC '05), pages 269–282, 2005.


ness, P. F. Patel-Schneider, and L. A. Stein. OWL web ontology language 

1998.


service for corba. In CDS ’98: Proceedings of the International Conference 
IEEE Computer Society.

ABLE: A toolkit for building multiagent autonomic systems. IBM Systems 


[22] M. Boesen and J. Madsen. eDNA: A bio-inspired reconfigurable hardware 
cell architecture supporting self-organisation and self-healing. In Conference 
on Adaptive Hardware and Systems (AHS ’09), pages 147–154, August 2009.


Index

activation & control engine, 136
  bootstrapping, 139
communication, 137, 138
  synchronisation, 138
autonomic computing, 3
  blueprint architecture, 4
  control theory, 3
  feedback loop, 3
autonomic element, 11
autonomic manager, 4
autonomic system, 11
  dynamically structured, 14
  statically structured, 15
unstructured, 12
behavioural complexity, 24
contextual knowledge, 28
distributed system, 2
  complexity, 2
  management, 2
information system, 1
instrumentation, 140
  AOP, 141
  sensor value provider, 142
managed resource, 4
management model, 46
  analyser, 49
  AnalyserStrategicRules, 49
  SymptomOccurrenceRules, 49
autonomic manager, 59
  AutonomicProcessResult, 64
diagnoser, 51
  ChildResultToSymptomRules, 53
  DiagnoserStrategicRules, 53
  DiagnosisDeterminationRules, 53
  HypothesisEvaluationRules, 53
  HypothesisSelectionRules, 52
  HypothesisValidationRules, 53
information flow entity, 64
  Diagnosis, 68
  Effector, 72
  Hypothesis, 68
  Plan, 69
  Sensor, 64
  Symptom, 67
plan translator, 58
  PlanTranslationRules, 58
  TranslatorStrategicRules, 58
planner, 56
  PlannerStrategicRules, 56
  PlanSelectionRules, 56
OWL, 98
  cardinality restriction, 99
  class, 98
  existential restriction, 99
  individual, 98
  property, 98
plan, 69
  actions constructs, 70
  initial plan, 69
  inspective plan, 70
  remedy plan, 70
research approach
  action research, 7
  design science, 7
  interpretivism, 6
  positivism, 6
rule engine, 139
self-management knowledge, 95
self-management ontology, 100
semantic web, 97
software fault handling process, 32
structural complexity, 25
SWRL, 98
  example generic rule, 117, 129
  example specific rule, 118, 130
system behaviour, 2, 30
  macroscopic, 17
  microscopic, 17
system model, 81
  behavioural model, 81
    Event, 86
    Job, 82
    State, 85
    Task, 83
structural model, 81
  ManagedClass, 92
  ManagedComponent, 91
  ManagedConnector, 90
  ManagedMethod, 92
  ManagedRunnable, 88
  ManagedSystem, 88
  Protocol, 90

unit of management, 26
  behavioural perspective, 27
  structural perspective, 27
use-case, 30
  characteristics, 31
  extended definition, 31
  levels, 32
    class level, 33
    component level, 33
    runnable level, 33
    system level, 33
  references, 34
    horizontal reference, 34
    vertical reference, 35
Curriculum Vitae

Reza Haydarlou was born in Khoy (Iran) on October 16, 1961. He completed his pre-university education in Iran in 1979. In January 1988, he moved to The Netherlands. After passing some admission examinations, he started his study in Computer Science at the University of Amsterdam (UVA) in September 1988. In 1993, he obtained his M.Sc. in Computer Architecture. After his graduation, he worked five years as a System Programmer for a company supplying hospital information systems. In September 1998, he changed his job and started to work in a banking enterprise. Alongside his work as an Architect, in January 2004, he started his Ph.D. trajectory under the supervision of Prof. dr. F.M.T. Brazier in the Intelligent Interactive Distributed Systems (IIDS) group, Faculty of Sciences (Department of Computer Science) of the Vrije Universiteit Amsterdam (VU). In September 2009, the IIDS group moved to the Systems Engineering Section of the Technology, Policy and Management Faculty of Delft University of Technology (TUDelft). For his Ph.D. trajectory, he spent two days a week from 2004 to 2008, and one day a week from 2008 till now. During his Ph.D. trajectory, Reza presented his research at several international conferences in China, Hong Kong, Greece, the Netherlands, and Denmark. He also published his work in several academic proceedings.