HAL
open science

Cube: a decentralised architecture-based framework for software self-management

Bassem Debbabi

To cite this version:

Bassem Debbabi. Cube: a decentralised architecture-based framework for software self-management. Software Engineering [cs.SE]. Université de Grenoble, 2014. English. NNT : . tel-00951532

HAL Id: tel-00951532

https://theses.hal.science/tel-00951532

Submitted on 24 Feb 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 7 août 2006

Présentée par

Bassem DEBBABI

Thèse dirigée par **Philippe LALANDA** et
codirigée par **Ada DIACONESCU**

préparée au sein du **Laboratoire d'Informatique de Grenoble**
dans **'École Doctorale Mathématique, Science et Technologies de l'Information, Informatique (MSTII)**

Cube: a decentralised architecture-based framework for software self-management

Thèse soutenue publiquement le **28 janvier 2014**,
devant le jury composé de :

M. Ioannis PARISSIS
Professeur à Grenoble INP – Esisar Valence, Président
M. Giuseppe VALETTO
Professeur, Fondazione Bruno Kessler, Trento Italy, Rapporteur
M. François TAIANI
Professeur à l'Université de Rennes 1, Rapporteur
Mme. Salima HASSAS
Professeure à l'Université Claude Bernard Lyon 1, Examinatrice
Mme. Catherine HAMON
Ingénieur de recherche à Orange Labs Meylan, Invitée
M. Philippe LALANDA
Professeur à l'Université de Grenoble, Directeur de thèse
Mme. Ada DIACONESCU
Maitre de Conférences à Télécom ParisTech, Co-directrice de thèse

Acknowledgments

First and foremost, I would like to sincerely thank my advisors, Prof. Philippe Lalanda and Dr. Ada Diaconescu, for guiding me at every step throughout my Ph.D, for everything I have learned about research because of them, and for their patient guidance, candid advice, kind encouragement, and for their sense of humour.

I would like to thank Prof. Giuseppe Valetto and Prof. François Taiani for spending time to carefully review my thesis. Their comments and suggestions helped me to strengthen and elevate the quality of my thesis. I would also like to thank my other thesis committee members, Prof. Ioannis Parissis and Prof. Salima Hassas for examining my thesis.

And, a man cannot long and happily live without his friends. Many thanks to Abdessalem Chekhchoukh, Fahim Maache, Hilal Goutas, Moussa Brahimi, Halim Mouellef, Abdelgani Sam, and Mehdi Rafik Benmaissa. I would also like to thank all the past and present ADELE research team members for the good atmosphere they created on the second floor of the C building. I thank especially all those with whom I worked among others Issac, Elmehdi, Gabriel, Clément, Stéphanie, and Jonathan; and those with whom I had long discussions at the cafète.

I would like to express my sincere gratitude to my wife Fouzia for here continued support and patience, and my sincere appreciation to my father, mother, sisters and brother, and all my family in Algeria for supporting and understanding me.

Last, but not least, I would like to dedicate this thesis to my daughter (not yet born at the time of my thesis defence). I hope for her and for all my future children all possible success in their lives.

شكرا Merci Thanks Gracias 谢谢

باسم دبّابي

Bassem Debbabi

Abstract

In recent years, the world has witnessed the rapid emergence of several novel technologies and computing environments, including cloud computing, ubiquitous computing and sensor networks. These environments have been rapidly capitalised upon for building new types of applications, and bringing added-value to users. At the same time, the resulting applications have been raising a number of new significant challenges, mainly related to system design, deployment and life-cycle management during runtime. Such challenges stem from the very nature of these novel environments, characterized by **large scales**, high **distribution**, resource **heterogeneity** and increased **dynamism**.

The main objective of this thesis is to provide a generic, reusable and extensible self-management solution for these types of applications, in order to help alleviate this stringent problem. We are particularly interested in providing support for the runtime management of system architecture and life-cycle, focusing on applications that are component-based and that run in highly dynamic, distributed and large-scale environments. In order to achieve this goal, we propose a **synergistic solution – the Cube framework** – that combines techniques from several adjacent research domains, including self-organization, constraint satisfaction, self-adaptation and self-reflection based on architectural models.

In this solution, a set of **decentralised Autonomic Managers** self-organize dynamically, in order to build and administer a target application, by following a **shared** description of administrative goals. This formal description, called **Archetype**, contains a graph-oriented specification of the application elements to manage and of various constraints associated with these elements. A prototype of the Cube framework has been implemented for the particular application domain of **data-mediation**. Experiments have been carried-out in the context of two national research projects: Self-XL and Medical. Obtained results indicate the viability of the proposed solution for creating, repairing and adapting component-based applications running in distributed volatile and evolving environments.

Keywords: *Autonomic Computing, Software Architecture, Framework, Software Engineering, Self-Management, Self-Adaptation, Self-Organisation.*

Résumé

Durant ces dernières années, nous avons assisté à une forte émergence de nouvelles technologies et environnements informatiques tels que le cloud computing, l'informatique ubiquitaire ou les réseaux de capteurs. Ces environnements ont permis d'élaborer de nouveaux types d'applications avec une forte valeur ajoutée pour les usagés. Néanmoins, ils ont aussi soulevés de nombreux défis liés notamment à la conception, au déploiement et à la gestion de cycle de vie des applications. Ceci est dû à la nature même de ces environnements **distribués**, caractérisés par une grande **flexibilité**, un **dynamisme** accru et une forte **hétérogénéité** des ressources.

L'objectif principal de cette thèse est de fournir une solution générique, réutilisable et extensible pour l'auto-gestion de ces applications. Nous nous sommes concentrés sur la fourniture d'un support logiciel permettant de gérer à l'exécution les architectures et leur cycle de vie, notamment pour les applications à base de composants s'exécutant dans des environnements dynamiques, distributes et à grande échelle. De façon à atteindre cet objectif, nous proposons une **solution synergique – le framework Cube –** combinant des techniques issues de domaines de recherche adjacents tels que l'auto-organisation, la satisfaction de contraintes, l'auto-adaptation et la reflexion fondée sur les modèles architecturaux.

Dans notre solution, un ensemble de **gestionnaires autonomiques décentralisés** s'auto-organise de façon à construire et gérer une application cible en s'appuyant sur une description partagée des buts de l'application. Cette description formelle, appelé **Archetype**, prend la forme d'un graphe orienté exprimant les différents éléments de l'architecture et un ensemble de contraintes. Un prototype du framework Cube a été implanté dans le domaine particulier de **la médiation**. Des expériences ont été conduites dans le cadre de deux projets de recherché nationaux: Self-XL et Medical. Les résultats obtenus démontrent la validité de notre approche pour créer, réparer et adapter des applications à base de composants s'exécutant dans des environnements distribués, dynamiques et hétérogènes.

Mots-clés : *Informatique autonomique, Architecture logicielle, Framework, Génie Logiciel, Auto-gestion, Auto-adaptation, Auto-organisation.*

Contents

1 Introduction **1**
1.1 Importance and challenges of software management . . . 1
1.2 Cube approach . . . 3
1.3 Contributions of this thesis . . . 4
1.4 Scope and Applicability . . . 5
1.5 Organisation of this thesis . . . 5

2 Autonomic Computing **7**
2.1 Autonomic Computing Initiative . . . 7
2.1.1 IBM's perspective and vision . . . 7
2.1.2 Holistic approach . . . 9
2.1.3 Research communities . . . 10
2.2 Characteristics of autonomic computing systems . . . 11
2.3 Objectives of autonomic computing systems . . . 12
2.4 Reference architecture of autonomic computing . . . 13
2.4.1 Autonomic Element . . . 13
2.4.2 The MAPE-K loop . . . 15
2.4.3 Monitoring . . . 16
2.4.4 Analysis . . . 17
2.4.5 Planning and Execution . . . 19
2.5 Engineering autonomic systems . . . 19
2.5.1 Quality attributes . . . 20
2.5.2 Autonomic Frameworks . . . 21
2.5.3 Comparison . . . 28
2.6 Summary . . . 31

3 Approaches to Autonomic Computing **33**
3.1 Control Theory . . . 33
3.1.1 Types of Control Loops . . . 34
3.1.2 Related work to Autonomic Computing . . . 35
3.1.3 Summary . . . 37
3.2 Agent-based systems . . . 37
3.2.1 Structural organisation of multi-agent systems . . . 38
3.2.2 Communication principles . . . 39
3.2.3 Adaptive multi-agent systems . . . 40
3.2.4 Research work related to Autonomic Computing . . . 41

3.2.5 Summary . . . 42
3.3 Nature-inspired approaches . . . 43
3.3.1 Self-organisation and emergence . . . 43
3.3.2 Functional structure and pattern formation . . . 45
3.3.3 Work related to Autonomic Computing . . . 45
3.4 Architecture-based approaches . . . 46
3.4.1 Software Architecture . . . 47
3.4.2 Adaptation concerns . . . 48
3.4.3 Enabling Techniques . . . 48
3.4.4 Summary of architecture-based approaches . . . 52
3.5 Discussion and chapter summary . . . 53

4 Proposition **55**
4.1 Problems addressed . . . 55
4.2 Contributions overview . . . 57
4.2.1 Positioning of the thesis within the team research . . . 57
4.2.2 Thesis contribution . . . 58
4.3 Objectives and Applicability . . . 59
4.3.1 Design decisions . . . 59
4.3.2 Applicability assumptions and scope . . . 60
4.4 Cube Framework overview . . . 62
4.4.1 Theoretical Support . . . 62
4.4.2 Solution Overview . . . 63
4.4.3 Coordination Strategy . . . 64
4.4.4 Cube Autonomic Manager . . . 68
4.4.5 System Management Life-Cycle . . . 69
4.5 Summary . . . 71

5 Cube Framework **73**
5.1 Goal and Knowledge Specifications . . . 74
5.1.1 Overview . . . 74
5.1.2 Key model-based management concepts . . . 75
5.2 System Modelling Language – SML . . . 76
5.2.1 ECORE . . . 77
5.2.2 Cube Meta-Model . . . 78
5.2.3 Domain-Specific Managed Elements . . . 80
5.2.4 Cube Runtime Model . . . 82
5.3 Goal Description Language - GDL . . . 83
5.3.1 GDL Elements . . . 85
5.3.2 GDL Properties . . . 85
5.3.3 Archetype . . . 87
5.4 Cube Autonomic Manager . . . 89
5.4.1 Internal Architecture Overview . . . 89
5.4.2 Runtime Model Controller . . . 90
5.4.3 Archetype Resolver . . . 92
5.4.4 Life Controller . . . 96
5.4.5 Monitors/Executors (Technology-Specific Extensions) . . . 98

5.4.6 Communicator . . . 99
5.5 Summary . . . 101

6 Implementation **103**
6.1 Overview . . . 104
6.1.1 Supporting Technologies . . . 104
6.1.1.1 OSGi . . . 104
6.1.1.2 Apache Felix iPOJO . . . 107
6.1.2 Execution Platform . . . 109
6.2 XML Syntax for the Archetype model . . . 110
6.2.1 The Archetype Document . . . 110
6.2.2 Archetype Elements and Description Properties . . . 111
6.2.3 Archetype Goals . . . 113
6.2.4 Complete Example . . . 114
6.2.5 Archetype Parser . . . 115
6.3 Cube Execution Platform . . . 117
6.3.1 Overall Architecture . . . 117
6.3.2 Cube Runtime and its Administration Service . . . 117
6.3.3 Autonomic Manager . . . 119
6.3.4 Runtime Model Controller . . . 121
6.3.5 Data Structure and Implementation for the Archetype Resolver . . . 124
6.4 Extensions . . . 128
6.4.1 Overview . . . 128
6.4.2 Extension Points . . . 129
6.4.3 Deploying and using extensions . . . 134
6.4.4 Extensions provided by the Cube Framework . . . 135
6.5 Related Tools . . . 140
6.5.1 Overview . . . 140
6.5.2 Hot Deployer . . . 140
6.5.3 Console Commands . . . 141
6.5.4 Dynamic Web Monitoring Console . . . 142
6.6 Summary . . . 145

7 Evaluation **147**
7.1 Overview . . . 148
7.1.1 Introduction to Mediation . . . 148
7.1.2 Cilia Mediation Framework . . . 149
7.2 Case Study 1: monitoring the consumption of home resources . . . 152
7.2.1 Context . . . 152
7.2.2 Managed System . . . 152
7.2.3 Problems addressed . . . 154
7.2.4 Solution proposed . . . 155
7.2.4.1 Model Abstractions . . . 155
7.2.4.2 Extensions . . . 155
7.2.4.3 Archetype . . . 158
7.2.5 Scenarios and Results . . . 159
7.3 Case Study 2: health-care monitoring system . . . 167

7.3.1 Context . . . 167
7.3.2 Managed System . . . 168
7.3.3 Problems and Requirements to address . . . 169
7.3.4 Solution proposed . . . 170
7.3.4.1 Model Abstractions . . . 171
7.3.4.2 Extensions . . . 171
7.3.4.3 Archetype . . . 174
7.3.5 Scenarios and results . . . 178
7.4 Evaluation . . . 181
7.5 Summary . . . 185

8 Conclusion **187**
8.1 Summary . . . 187
8.2 Summary of thesis contributions . . . 189
8.3 Limitations and perspectives . . . 190
8.3.1 Archetype Specification . . . 190
8.3.2 Runtime Autonomic Management . . . 191
8.3.3 Towards a reusable methodology and integrated development environment for engineering autonomic computing systems . . . 195

A XML Terminology and Vocabulary **199**

B Implementation Details **201**
B.1 Administration Service . . . 201
B.2 Component modelled element . . . 202
B.3 Connected Specific Resolver . . . 204
B.4 Console Commands . . . 206

C Archetype XML files **207**

D Source code statistics **211**

Bibliography **213**

List of Figures

1.1 Increasing Complexity of Software over Time (adapted from [WH06]) 2

2.1 Autonomic element [LDJ13] 14
2.2 MAPE-K loop of Autonomic Computing Element 15
2.3 Technologies applied to the four stages of the autonomic manager (adapted from [DSNH10]) 15
2.4 Ganglia Monitoring System Report of Wikipedia Servers 17
2.5 Ceylon framework 21
2.6 Rainbow framework 23
2.7 Unity framework 24
2.8 AutoMate Autonomic Element [PLL+06] 25
2.9 AutoMate Composition Manager 26
2.10 Auto-Home Hierarchical Architecture 26
2.11 Encapsulating legacy systems with Jade 27
2.12 Jade autonomic framework 28

3.1 Feedback control system 35
3.2 Feedback and feed-forward control system 35
3.3 Architecture of Entropy 36
3.4 Computer System Agent in its environment 38
3.5 Example of agent organisations: a) hierarchical; b) team-based; c) federation. 39
3.6 Example of KQML message exchange 40
3.7 A taxonomy of coordination strategies [Wei99] 40
3.8 Decentralized control in self-organizing systems (adapted from [Dre07]) 44
3.9 Autonomic architecture inspired by the behaviour of ant colonies 46
3.10 Smart-home feature model at runtime. 50

4.1 Cube solution overview 64
4.2 Life-cycle of Cube-based autonomic management process 70

5.1 Overview of Cube's model-based management 76
5.2 Cube framework's Modelling Layers and Languages 77
5.3 ECORE Meta-model 78
5.4 Cube Meta-model 79
5.5 Cube Meta-model fragment and its ECORE model instance equivalent 80
5.6 Cube framework's Core Domain-Specific Managed Elements 80
5.7 Core Domain-Specific Model fragment and its corresponding Cube Meta-model instance . 81
5.8 Cube Runtime Model Example 83

5.9 GDL Example . . . 84
5.10 Examples of Description Properties in the GDL model . . . 86
5.11 Example of a Goal Property in the GDL model . . . 87
5.12 Cube framework's Autonomic Manager . . . 89
5.13 State Diagram of a Modelled Element Instance . . . 91
5.14 Cube AM's Archetype Resolver . . . 93
5.15 Example of a Constraint Satisfaction Problem (CSP) . . . 94
5.16 Cube framework's Life Controller . . . 97
5.17 Example of Cube framework's Life Controller . . . 97
5.18 Cube Monitors/Executors . . . 99
5.19 Cube Message Representation . . . 99

6.1 OSGi Platform . . . 104
6.2 OSGi Layers . . . 105
6.3 The Module Layer of the OSGi platform . . . 105
6.4 State diagram of an OSGi bundle's life cycle . . . 106
6.5 The Service Layer of the OSGi platform . . . 107
6.6 iPOJO Component-model . . . 108
6.7 Cube's Execution Platform . . . 109
6.8 Example of an Archetype document in a graph format . . . 114
6.9 Archetype XML Parser . . . 115
6.10 UML class diagram of the Cube Runtime Platform . . . 117
6.11 Cube Runtime Bundle . . . 117
6.12 The Autonomic Manager's Initial Configuration . . . 120
6.13 Starting Autonomic Manager Sequence Diagram . . . 120
6.14 Archetype Resolver Design . . . 124
6.15 Example of Local Archetype Resolver Resolution Graph. . . . 126
6.16 Example of distributed Archetype Resolver Resolution Graph . . . 127
6.17 Extension mechanism . . . 128
6.18 Abstract Managed Element class . . . 131
6.19 Adding new Monitors / Executors . . . 132
6.20 Adding a new Specific Resolver . . . 133
6.21 Adding new Communicator . . . 134
6.22 Cilia Monitors/Executors for the Cube framework . . . 138
6.23 Graphical Monitoring of the Runtime Model of one Autonomic Manager . . . 139
6.24 HotDeployer tool . . . 140
6.25 Cube Console . . . 141
6.26 Cube Console screen-shot . . . 142
6.27 Dynamic Web Monitoring Console Architecture . . . 143
6.28 Dynamic Web Monitoring Console Example. . . . 144

7.1 Mediation system . . . 148
7.2 Cilia Mediator component and its constituents . . . 150
7.3 Example of Cilia mediation chain . . . 151
7.4 Overview of a house resource monitoring application . . . 153
7.5 Distributed Cilia mediation chain . . . 154
7.6 Archetype part for Node self-management in Case Study 1 . . . 158

7.7 Self-creating and connecting Mediation components on home Gateways in case study 1 . . 159
7.8 Self-creating and connecting City and National Mediation components in case study 1 . . 160
7.9 Simplified instantiation solution for the archetype of case study 1 162
7.10 Number of control messages exchanged among Autonomic Managers 163
7.11 Average number of control messages for Scope formation and Archetype part resolution . 165
7.12 Average number of control messages for Scope formation and Archetype part resolution . 166
7.13 Data-mediation software for health-care service applications 168
7.14 Architecture of the mediation chain of "Health-care monitoring" use case 169
7.15 Overall mediation architecture with load-balancing between mediation servers 170
7.16 Cube Extensions for case study 2 174
7.17 Archetype part for use case 2: dynamic scope management 175
7.18 Archetype part for use case 2: self-creating components on the Aggregation Server 175
7.19 Archetype part for use case 2: self-creating components on each Mediation Server 176
7.20 Archetype part for use case 2: self-integrating new PHR providers 177
7.21 Archetype part for use case 2: self-removing mediation chains from inactive servers 177
7.22 Case study 2: load-balancing based on the average CPU load 179
7.23 Case study 2: self-healing of the mediation cluster 180

8.1 Actor roles 196

B.1 Administration Service Java Interface 201

List of Tables

2.1 Autonomic computing research communities 11
2.2 Comparison of control placement and organization 29
2.3 Comparison of specific/generic solutions and of self-* properties 30
2.4 Comparison of Quality Attributes 30

5.1 Graphical notation of GDL concepts 85

6.1 Autonomic Manager Configuration Entries 121
6.2 Extension Points of Cube Autonomic Manager 129
6.3 Manaed Elements contributed by the core Extension 135
6.4 ManagedElement-related Specific Resolvers 136
6.5 Component-related Specific Resolvers 136
6.6 Node related specific resolvers 137
6.7 Scope related specific resolvers 137

7.1 Case study 1: model abstractions and their semantics 155
7.2 Case study 1: used extensions and their possible configuration properties 156
7.3 Autonomic Managers for Case Study 1 160
7.4 Model abstractions for the second case study 172
7.5 Case study 2: extensions used and their configuration properties 172
7.6 Testing configuration for case study 2 178

B.1 Cube Console Commands 206

Chapter 1

Introduction

Contents

1.1 Importance and challenges of software management 1
1.2 Cube approach 3
1.3 Contributions of this thesis 4
1.4 Scope and Applicability 5
1.5 Organisation of this thesis 5

In this first chapter, we present the context of our work, the challenges, the adopted approach, the contributions of our thesis, as well as their scope of applicability. We also detail the structure of this document and the content of each chapter.

1.1 Importance and challenges of software management

Software systems have become a central part of a rapidly growing range of products and services from all sectors. Today, such systems are essential in many areas of our lives. We use them at home, for business, in hospitals, for banking, and so on. Unfortunately, engineering and managing such software systems is becoming increasingly complex and costly.

From an engineering point of view, the first concern with developing software systems used to be about finding the best algorithmic solution to perform specific operations. Over time, software was distributed across several physical machines to improve non-functional qualities like performance, scalability or availability. This led to another level of complexity; the focus shifted from the complexity of developing algorithms to the complexity of building distributed concurrent systems and of structuring large-scale systems like cloud [RCL09] and grid computing [BFH03]. Nowadays, software systems interact with other systems, with devices and with people – i.e. hence turning into socio-technical systems - distributed across large deployment areas, possibly all around the world.

Hence, software systems are increasingly distributed, heterogeneous, decentralized and interdependent, and they operate more and more in dynamic and often unpredictable environments [FGG+06]. This situation requires system architects and developers to make software systems more adaptive and context-aware (see Figure 1.1), with the drawback of increasing development and management complexity. From a system management and control point of view, which is the focus of our thesis, software systems are hence becoming increasingly complex and so more difficult and costly to develop and maintain. By

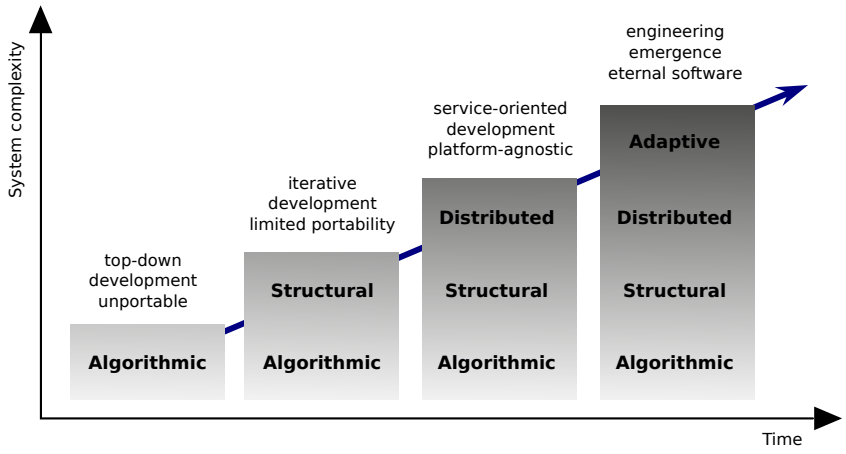


Figure 1.1: Increasing Complexity of Software over Time (adapted from [WH06])

management and control we mean administration operations that cover the entire computing eco-system, including computer hardware, network and software applications.

System administration is becoming more complex for several reasons. First, applications are often heterogeneous, consisting of several integrated sub-systems, like databases and application servers. An experimented operator is needed for each sub-system to maintain its correct and optimal execution. Experts of each technology must then communicate and share knowledge to meet global administration objectives. Second, many applications are distributed and so built on top of sophisticated middleware that has to be configured and re-configured at runtime. For instance, the use of EJBs [BMH06] may greatly simplify the developers' work but at the price of higher administration costs. Indeed, complicated declarative deployment descriptors have to be created at deployment-time and updated at runtime. Having to deal with heterogeneous middleware, by introducing middleware for middleware technologies, such as Web services, further complicates the situation. Third, the increasing scales of such systems, both in terms of number of components and area of deployment, amplifies the aforementioned challenges many times over. Last, but not least, systems pertaining to emerging domains like pervasive and ubiquitous computing [HMNS03] run in dynamic, heterogeneous and often unpredictable environments. In such cases, the system's execution context at deployment time is likely to change dramatically all along the application's lifetime. Corrective administration operations are required at runtime as a consequence.

Based on these considerations, we highlight the key system management challenges to be addressed in the following.

Resource heterogeneity. Administration has to deal with heterogeneous data, data-processing elements and communication support, from a variety of providers, based on diverse languages, and executed on different hardware and software platforms. Many parts are considered legacy systems, while others are more open and flexible. In-depth knowledge of the heterogeneous technologies involved is essential for administrating such systems efficiently.

Dynamicity. Many computing systems operate in highly-dynamic environments, hence experiencing frequent changes in both their internal constituents and their execution conditions. In pervasive computing for instance, devices can join or leave the system at any time. Some devices may be unknown to the system before their opportunistic integration at runtime. In all cases, the global software

system should continue its execution and cope with such dynamic phenomena without significant interruption.

Increasing number of integrated subsystems. Modern software systems allow the integration of several devices like smart-phones or home-boxes that include a variety of sensors and actuators. There has been a progressive shift from a centralized system view towards a strongly decentralized computing landscape. Namely, in more recent approaches, computation is distributed across a variety of relatively small devices, each one being responsible for specific tasks. The increasing number and distribution of such devices, combined with the heterogeneity and dynamicity issues identified above, make the administration of such applications progressively more difficult, at best. Consequently, management systems must be able to handle the increasing system complexity, in terms of scale, heterogeneity, distribution and dynamicity.

Increasing business requirements that change frequently during runtime. For most long-running applications, user requirements evolve over the system's lifetime. For example, non-functional properties specified initially may become inadequate and require adjustments to meet new business needs. Changes in user requirements may affect the entire software system and can be extremely complex to handle correctly and efficiently.

Autonomic Computing [Hor01] proposes to address such challenges by automating some of the system management tasks. Nonetheless, the automatic administration of complex computing systems requires that the autonomic management applications are also complex and capable to adapt dynamically. In this sense, autonomic management applications absorb the complexity involved in system administration, thus hiding it from human managers. Human administrators can interact with the autonomic system via high-level policies and via concepts pertaining to their business domain. At the same time, to achieve such policies and goals automatically, the additional complexity must now be handled by the software engineers that develop and maintain the autonomic system. This task is highly challenging and costly in itself, placing important requirements on the design of such complex autonomic systems. It also indicates that existing frameworks fall short of addressing some of the important requirements currently challenging autonomic system developers and administrators.

1.2 Cube approach

Cube [DL09, DL11, DDL12] is a generic approach for designing autonomic management systems that can address the aforementioned challenges – i.e. heterogeneity, dynamic adaptability, scalability and evolution of business objectives. It proposes a solution that mixes two well-known approaches: totally decentralised, bottom-up approaches and completely deterministic, top-down approaches. "Bottom-up" approaches promote highly-decentralized control logic, based on fully distributed and decoupled controllers. Its main advantages include general survivability and scalability in terms of data-processing power. Nonetheless, the difficulty of predicting the overall behaviour of a decentralised system, along with possible emergent phenomena, makes the design and maintenance of such systems rather complicated and costly. Moreover, the additional communication required for coordinating the various system parts may have a negative impact on system scalability. Conversely, "top-down" approaches are based on centralised, highly-deterministic control logic. Here, the decision-making process relies on a global system representation and can plan coherent actions on managed software components, hence ensuring more easily the correctness and effectiveness of adaptation and self-management operations.

The Cube approach aims to combine the two aforementioned approaches in order to capitalise on their

advantages while avoiding their respective shortcomings. Namely, Cube relies on decentralized decision-making modules coordinated via well-specified protocols. This allows the system to take local decisions more rapidly based on local system views. It avoids having to maintain global system representations and to consider them entirely when taking each management decision. Hence, resource consumption is distributed across the decentralised processes, which is essential when managing large-scale systems requiring frequent adaptations. At the same time, the global self-organising behaviour of a Cube autonomic system is controlled via a formal goal specification, which is replicated across all decentralised processes. This ensures that the overall computing system managed via a decentralised approach will always meet the constraints in the goal specification. This level of determinism is a typical advantage of a centralised approach.

More precisely, the Cube approach promotes two core techniques:

- Top-down: use of an *abstract architectural model* (or *archetype*) to formally specify administrative goals; this aims to control the autonomic management process and ensure the viability of resulting system configurations;
- Bottom-up: *decentralisation of the autonomic management* process, in order to avoid a single point of control and the associated limitations to robustness and scalability; decentralised managers coordinate to obtain coherent systems that conform to the architectural model.

While this approach seems promising, it also raises difficult problems that must be addressed when implementing concrete autonomic systems. The most important questions include:

- How to design and formalise the architectural model? The model must be restrictive enough to ensure core properties of the managed system while also allowing sufficient variability for adaptation;
- How to split administrative tasks among decentralised autonomic managers? Dynamic changes in the managed system impose that task assignation to autonomic managers be adaptable during runtime;
- How to coordinate the activities of autonomic managers so as to ensure the coherence and conformance of the resulting system to the modelled objectives? Autonomic managers may have to coordinate both locally and remotely, to ensure local or more global system properties, respectively. - How to ensure the extensibility of autonomic managers to support future business objectives and various types of managed resources?

1.3 Contributions of this thesis

The contribution of this thesis is positioned within the general Cube project. It consists of a concrete framework for facilitating the development of autonomic applications following the Cube approach. Notably, the proposed Cube framework provides concrete, viable solutions to the difficult problems raised by the general Cube approach (as listed above). Moreover, the thesis provides a concrete framework extension customised for a specific application domain – distributed data-mediation.

More generally, this thesis developed a solution for overcoming the limitations identified in the state-of-art for engineering autonomic computing systems. Our main contributions consist of:

- Providing an **autonomic framework** – called Cube framework - that features several important quality attributes and is characterised by:
    - A high-level, extensible, graph-oriented *language* for specifying administration objectives in a formal manner;
    - A decentralised, modular and extensible *runtime* platform that ensures the specified objectives.

- Identifying and applying several **techniques** in a synergistic manner in order to ensure important quality attributes in autonomic management. These techniques include: *architecture-oriented modelling*, *constraint resolution*, *self-organisation* and *control theory*;
- A working **open-source tool set** that integrates: a lightweight, dynamically adaptable runtime platform based on standards; a modular framework design and implementation enabling the seamless development and integration of customised domain-specific extensions; and, a set of common extensions and tools helping the debugging and monitoring of Cube systems.
- A concrete **framework application to distributed data-mediation systems** for self-managing the instantiation and configuration of mediation components, their local and remote interconnections, the self-healing and self-extension of distributed mediation chains, and the self-optimisation of resource consumptions.

1.4 Scope and Applicability

Although in this thesis we aimed to cover a broad range of modern software systems, the scope of our work is limited currently to autonomic systems based on component-oriented designs, which are quite widely used in modern computing systems. In addition, we limit our initial scope to systems where all managed components are stateless. This means that updating a component will not require the migration of its existing state to the new component instance. Finally, we assumed that the autonomic system has access to a well-connected communication network. In general, we only concentrate on the overall architecture of the application, and leave low-level details for future research. The experiments that support our contributions were conducted in the context of two national research projects, which made use of data-mediation systems [Wie92, WG97]. In both cases, the data-mediation applications administered via the Cube frameworks were implemented based on the Cilia mediation framework [Deb09, GPD+10, GMD+11]. The first research project – SelfXL [1] - focused on the self-management of complex large-scale systems in general; the second project – Medical [2] - is a regional research and development project aiming to provide a solution for medical assistance at home.

1.5 Organisation of this thesis

The remainder of this thesis is organised as follow:

- Chapter 2 introduces the Autonomic Computing initiative, its objectives, its reference architecture for autonomic systems, and a representative list of existing frameworks.
- Chapter 3 details some of the significant domains from which we have adopted concepts and techniques for developing our framework. This includes control theory, agent-based systems, nature-inspired approaches and architecture-based management solutions.
- Chapter 4 introduces our proposal, its objectives and applicability, and provides an overall description of the developed framework.
- Chapter 5 details the proposed framework, including the System Modelling Language SML, Goal Description Language GDL (also called Archetype Language) and the internal architecture of a Cube Autonomic Manager.

[1] http://selfxl.forge.inria.fr
[2] http://medical.imag.fr

- Chapter 6 provides more implementation details for our framework. In particular, it details the XML syntax for the Goal Description Language used to describe administrative goals, the Runtime Execution Platform, the extension mechanisms and supported extension points, and some of the related tools for helping Cube developers and administrators.
- Chapter 7 details the evaluation process and the results obtained by experimenting with the Cube framework in the context of two national projects. It indicates the viability of our proposal as highlighted by these experiments.
- Chapter 8 concludes our thesis by providing a global synthesis of our contributions and discussing the current limitations and perspectives.

Chapter 2

Autonomic Computing

Contents

2.1 Autonomic Computing Initiative **7**
2.1.1 IBM's perspective and vision 7
2.1.2 Holistic approach 9
2.1.3 Research communities 10
2.2 Characteristics of autonomic computing systems **11**
2.3 Objectives of autonomic computing systems **12**
2.4 Reference architecture of autonomic computing **13**
2.4.1 Autonomic Element 13
2.4.2 The MAPE-K loop 15
2.4.3 Monitoring 16
2.4.4 Analysis 17
2.4.5 Planning and Execution 19
2.5 Engineering autonomic systems **19**
2.5.1 Quality attributes 20
2.5.2 Autonomic Frameworks 21
2.5.3 Comparison 28
2.6 Summary **31**

In the introductory chapter, we have seen that modern software systems are becoming increasingly complex to design, code, deploy and manage. In this chapter, we focus on an initiative called Autonomic Computing, which seeks to render computing systems as self-managed. After highlighting the vision and the perspectives of this initiative, we focus on the autonomic computing systems and their characteristics. Then, we study and compare the existing autonomic computing frameworks taking a software engineering point of view.

2.1 Autonomic Computing Initiative

2.1.1 IBM's perspective and vision

In 2001, Paul Horn - IBM's Senior Vice President of Research - introduced the Autonomic Computing Initiative in response to the ever growing complexity of integrating, managing, and operating computing

systems. The ultimate goal of this initiative was to be able to develop self-managing systems, hiding their complexity from administrators.

In both a keynote presentation, addressed to the National Academy of Engineers at Harvard University in October 2001, and an accompanying publication [Hor01], Paul Horn alerted the industrial and the academic communities about a looming crisis arising from the ever-increasing complexity of managing computing systems. He qualified this issue as "*our next Grand Challenge*". While there has been incredible progress in hardware (Moore's Law is still verified today), software systems are more and more difficult to develop and manage. They are often coded by thousands of developers, used by millions of clients over the Internet, and maintained by a significant number of qualified IT professionals.

While user needs for computing services are increasing, the demand for qualified people to keep the computing systems running is also growing. Unfortunately, over the long-term, simply increasing the number of qualified people is not a viable solution to this ever-growing complexity. This can be compared to what happened in the US telephony industry in the 1920s [Hor01]. At that time, in spite of the limited number of human operators working on manual switchboards the use of phones had exploded. This raised serious concerns that there would not be enough trained operators to work the switchboards [Mai02]. Without the development of automatic branch exchanges, analysts estimated that half the US population would have to work as a telephone operator if growth continued at the same rate.

Drawing from the analogy between the management requirements of the telephony and computing domains, Paul Horn pointed out that new approaches should be found in order to face the increasing IT complexity and to limit its impact on our daily operations. He suggested drawing inspiration from the human body and the way in which it manages its massively complex systems. More precisely, the Autonomic Nervous System (ANS) controls many of the body's internal organs. It ensures - without requiring our consciousness - those complex biological systems such as vision, digestion, respiration, or blood circulation work in harmony to maintain a steady internal state called homeostasis. In response to variations such as external temperature, posture, stress or food intake, the ANS adjusts our internal body functions most often without reaching our consciousness. It triggers largely automatic or reflex responses through the autonomic efferent nerves, thereby eliciting appropriate reactions of the heart, vascular system and other complex biological systems.

Without the ANS, we would be constantly occupied with adapting our body's vital functions so as to maintain homeostasis in varying environmental conditions. Accordingly, IBM suggested that computing systems should also have autonomic behaviours as observed in the Autonomic Nervous System (ANS). Complex computing systems should manage themselves with limited or no human intervention. They should adjust their internal functions in order to address varying circumstances and conditions, hence freeing administrators from the inherent complexity and cost of management tasks. When required, human intervention should be made available via high-level interfaces, based on high-level policies and business-domain concepts rather than on low-level technical configurations.

Two years after Paul Horn's initiative, Jeffrey Kephart and David Chess from IBM's Thomas J. Watson Research Center promoted a vision and roadmap for the newly created community around the Autonomic Computing initiative. They published "*The Vision of Autonomic Computing*" [KC03] where they described some of the self-management capabilities that Autonomic Computing Systems should exhibit in order to achieve high-level objectives specified by administrators. This includes *self-configuration, self-optimization*, *self-healing*, and *self-protection*. They suggested an accompanying conceptual architecture consisting of a collection of cooperating autonomic elements, and laid out a number of key engineering and science research challenges that should be addressed by multiple efforts of experts from different fields. This vision influenced a significant number of research papers[1], many of which addressed the way

[1]More than 3500 direct citations according to Google Scholar

of realizing the autonomic computing vision [DSNH10] by exploring techniques from related disciplines, like artificial intelligence, neural networks, complex adaptive systems, catastrophe theory, cybernetics, self-evolving systems, heterogeneous workload management and control theory.

2.1.2 Holistic approach

The Autonomic Computing approach promotes a global, systematic approach that allows the integration and coordination of multiple computing systems and their self-management as a whole [Hor01, KC03]. As we will see in greater details in this thesis, Autonomic Computing covers a wide range of tools and techniques, including execution platforms, programming models and specialized algorithms, to allow the self-management of complex computing systems. An important challenge, of course, is to integrate these techniques in order to provide replicable solutions.

This holistic approach appears in contrast to many existing computing systems that are specialized in a particular management function, such as optimizing a few attributes or functionalities (e.g., a database query optimizer). In the autonomic case, the challenge is more important since it seeks to render complete systems as self-managed. For instance, an autonomic upgrade facility for accounting systems [KC03] must deploy several autonomic elements to control the different phases of the upgrade process. Also, a Grid Computing system [BFH03] spans several heterogeneous distributed computers. Here, autonomic elements have to be deployed on each node to control the state and the context of that node, analyse its requirements (fault tolerance, performance, QoS, security, etc.), and adapt it to satisfy administrative objectives [PH06].

From another point of view, Autonomic Computing is seen as an attempt to merge a selection of existing fields with the aim of building self-managing systems. Autonomic Computing does not specify what types of technologies should be used to achieve its goals, but any existing technology that exhibits behaviours or partial behaviours of pervasiveness and self-management can be classified as Autonomic Computing [LML05]. Huebscher and McCann define Autonomic Computing as "*a concept that brings together many fields of computing with the purpose of creating computing systems that self-manage*" [HM08]. In the same direction, Muller and colleagues define it as "*not a new field but rather an amalgamation of selected theories and practices from several existing areas*" [MMWW06]. Sterritt and colleagues argue that Autonomic Computing "*is emerging as a significant new strategic and holistic approach to the design of complex distributed computer systems*" [SPTU05]. Accordingly, "*what is new is Autonomic Computing's holistic aim of bringing all the relevant areas together to create a change in the industry's direction: self-ware instead of the hardware and software feature upgrade cycle of the past that created the complexity and total cost of ownership quagmire*" [Ste05].

Autonomic Computing draws inspiration from two major high-level domains: the study of complex natural systems and the development of highly-complicated computing systems. Relevant computing systems have been developed in highly diverse domains ranging from traditional automation systems to systems based on Artificial Intelligence techniques. They provide algorithms, architectures, models and techniques that can be directly reused in Autonomic Computing. In turn, natural systems are not software, their study can include domains as diverse as Economics, Biology, Chemistry or Physics, but they can also provide contributions to Autonomic Computing in terms of theories and models.

Specifically, autonomic system designers can adopt results from the following domains:

- Automatic control. This domain includes Control Theory and Control Engineering applications, which have been applied to mechanical, electrical, chemical or financial systems.
- Robotics. Robots often operate in highly dynamic environments where adaptation abilities are essential and opportunities for human intervention are limited or undesirable.

- Multi-Agent Systems. The key strength of these systems lies in their capacity to address complicated computing problems by dividing and distributing them amongst a set of specialized reasoning agents.
- Software Engineering. Many paradigms and technologies have been introduced for ensuring application robustness and flexibility in execution contexts susceptible to frequent (dynamic) change.

These domains provide an essential base for developing autonomic computing systems. They are presented in more details in the next chapter, in the light if our objectives and approach. From a domain-specific perspective, several IT fields have been striving to address challenges that are similar to those confronting autonomic systems [LDJ13]. These include pervasive and ubiquitous systems, smart grids, self-adaptive and context-aware applications and middleware, dependable computing systems, and so on.

The interrelations among concepts like self-adaptive, self-organizing and autonomic systems can often cause a certain degree of confusion. There are similarities and differences between these concepts. Self-adaptation enables a software system to modify its structure and behaviour dynamically in response to changes in the execution environment [MSKC04]. Works in self-adaptation often focus on the application and middleware layers, while autonomic computing also covers lower layers like the operating systems and the network. Self-organization enables systems consisting of a large number of subsystems to perform a collective task [Dre07]. Today, self-organization is mostly applied to low-level communication protocols and services, whereas the autonomic computing initiative covers large-scale systems, as well as centric systems. The concepts of these domains are however strongly related and in many cases can be interchangeably used [ST09].

In any case, a prominent feature of autonomic computing is that it takes a *holistic* approach to the design and development of computing systems that can adapt themselves to changing conditions. In our thesis, we use the term autonomic computing to cover all the other related domains. We only refer to each specific domain when talking in detail about the underlying techniques used to exhibit specific autonomic computing capabilities.

2.1.3 Research communities

Since its inception in 2001, Autonomic Computing has become an important field of research of its own, with a number of dedicated journals, conferences, and workshops. The term autonomic is often cited in other domains like Artificial Intelligence or Software Engineering. Many conferences in various computing domains include a special track on autonomic computing.

Table 2.1 presents some of the major publications and events created based on IBM's initiative. This list includes the IEEE International Conference on Autonomic Computing (ICAC), which is now in its ninth year, and is considered as the leading conference on autonomic computing techniques, foundations, and applications. The list also includes interdisciplinary communities, like SASO and SEAMS. The International Conference on Self-Adaptive and Self-Organizing Systems (SASO) aims to provide a forum for researchers with different backgrounds (e.g., complex systems, control theory, artificial intelligence, sociology and biology) to find new ways of designing and managing networks, systems and services. The International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS) concentrates more on software engineering-oriented approaches and trends, so as to better deal with the self-management.

Table 2.1: Autonomic computing research communities

Title	Acronym	Affiliation	Type
ACM Transactions on Autonomous and Adaptive Systems	TAAS	ACM	Journal
The Journal of Autonomic and Trusted Computing	JoATC	American Scientific Publishers	Journal
International Conference on Autonomic Computing	ICAC	IEEE	Conference
International Conference on Self-Adaptive and Self-Organizing Systems - 2007 until now 2012. (Merger of Self-Star, SelfMan and ESOA)	SASO	IEEE	Conference
International Symposium on Software Engineering for Adaptive and Self-Managing Systems	SEAMS	ACM	Workshop
International Workshop on Self-* Properties in Complex Information Systems – 2004	Self-Star	IEEE	Workshop
International Workshop on Self-Managed Networks, Systems & Services - 2005, 2006	SelfMan	IEEE	Workshop
The Autonomic Computing Workshop	AMS		Workshop
ACM Workshop on Self-Managed Systems	WOSS	ACM	Workshop

2.2 Characteristics of autonomic computing systems

When Paul Horn initially introduced the *Autonomic Computing Initiativ*e, he suggested eight general elements or characteristics that software systems should possess to be considered as *autonomic* [Hor01]. These characteristics are the following:

1. To be autonomic, a computing system needs to "know itself"— and comprise components that also possess a system identity. That is, in order to manage themselves, autonomic systems should have an elaborate knowledge of their current internal status and the interconnections with other systems and resources.
2. An autonomic computing system must configure and reconfigure itself under varying and unpredictable conditions. As the autonomic system can have hundreds of possible configurations, it must choose the best one for a context and do so within a reasonable amount of time.
3. An autonomic computing system never settles for the status quo — it always looks for ways to optimize its workings. It must monitor its metrics to self-optimize its computing and attain changing user objectives using adaptive algorithms, as well as advanced feedback control systems.
4. An autonomic computing system must perform something akin to healing — it must be able to recover from routine and extraordinary events that might cause some of its parts to malfunction. When errors occur, the autonomic system must be able to discover problems, and then find solutions to keep it functioning smoothly. Initially, such self-healing responses will follow rules given by human experts, but over the time will be supplemented by self-learning processes inherent to an autonomic system.
5. A virtual world is no less dangerous than the physical one, so an autonomic computing system must be an expert in self-protection. It must be able to detect and treat threads from viruses and intrusions by hackers to maintain overall system security and integrity.

6. An autonomic computing system knows its environment and the context surrounding its activ
7. An autonomic computing system cannot exist in a hermetic environment. Open standards for system identification, communication and negotiation are needed because autonomic systems run in heterogeneous and rapidly evolving computing environments.
8. Perhaps most critical for the user, an autonomic computing system will anticipate the optimized resources needed while keeping its complexity hidden. This is the ultimate goal for autonomic computing. It must carry out its different functions while keeping its complexity hidden from users.

2.3 Objectives of autonomic computing systems

These eight key characteristics of autonomic computing systems were further synthesized by Kephart and Chess [KC03] in terms of user objectives into four main properties. Each self-property describes a user objective or a "*quality requirement of autonomic system*" [LML05] that has to be considered when developing an autonomic system:

- **Self-Configuration**. An autonomic computing system configures itself according to high-level goals representing business-level objectives. That is, it adapts itself automatically to dynamically changing environments by adding/removing computer machines, installing/deleting software modules, setting up new configurations, and so on. For example, when a new software module is incorporated within a business application, it automatically integrates itself with the rest of the system, including needed configurations and connections to databases, web applications, and other instances of the same module. Accordingly, the system self-configures seamlessly and without interruption of service.
- **Self-Optimization**. An autonomic computing system optimizes its use of resources and performs optimally. It will continually monitor and tune itself automatically to improve its operation. These kinds of capabilities should be extended across various heterogeneous system constituents. Consequently, tuning a subsystem can have unanticipated effects on the entire system.
- **Self-Healing**. An autonomic computing system detects, analyses, diagnoses and fixes problems resulting from failures in hardware (e.g., hard disk problem) or software (e.g., bugs in a software component). It must be able to detect, fix or isolate the failed element, and try to find solutions for the problem without any apparent system disruption. It is important that the self-healing process does not introduce new bugs or the loss of vital system settings and functions.
- **Self-Protection**. An autonomic computing system protects itself from malicious attacks, user misuse and inadvertent software changes that can produce cascading failures. It automatically tunes itself to protect the system as a whole against the arising security problems. It can also anticipate security breaches by exhibiting proactive analysis.

These four initial self-* properties were provided for the specific context of enterprise system management. This list has grown substantially [BJM+05b] to include, among others, self-organizing, self-growing, self-diagnosis, self-adapting, self-destructing, self-reflecting, self-stabilizing, and so on.

These additional self-* properties that are now encountered in the literature can be defined as follows:

- Self-organizing: a system's ability of being automatically formed via the decentralised assembly of multiple independent elements, which become the system's constituent elements.
- Self-growing: a system's ability of expanding its internal structure depending on the context and its evolutions.
- Self-diagnosis: a system's ability to analyse itself in order to identify existing problems or to anti-

cipate potential issues.

- Self-adapting: a system's ability to modify itself (self-adjust) in reaction to changes in its execution context or external environment, in order to continue to meet its business objectives despite such changes.
- Self-destructing: a system's embedded capability to destroy itself, either because it determines that it is no longer capable of reaching its goals (e.g. a corrupted system shuts itself down in order to prevent affecting user safety or infecting neighbouring systems); or because it has reached a predefined expiration date.
- Self-reflecting: a system's ability to determine whether or not its self-* functionalities conform to expected operation. This may involve self-simulation operations. Within an autonomic system, self-reflection may be considered or implemented as a higher autonomic management layer (meta-management) that supervises and adapts the activities of the basic autonomic management layer, which supervises and adapts the managed system resources.
- Self-stabilizing: a system's ability to attain a stable, legitimate state, starting from an arbitrary state and after a finite number of execution steps. This property has been traditionally linked to fault-tolerance in distributed systems [BKPH09], but is receiving increasing attention from the self-management system community (e.g. ensuring that self-repair or self-optimisation operations converge towards a system state that complies with high-level policies).

Different definitions for autonomic computing were given based on a selection of these self-* properties and in different application domains. Accordingly, "the vision of autonomic computing is to create self-management abilities through self-* properties" [SH05].

2.4 Reference architecture of autonomic computing

2.4.1 Autonomic Element

To fulfil the autonomic computing vision, IBM researchers have established an architectural framework for autonomic systems [KC03] and an accompanying blueprint [IBM03]. This blueprint organizes an autonomic computing system into building blocks that can be composed together to bring self-* properties. These building blocks are referred to as autonomic elements.

As explained in [LDJ13], an autonomic system consists of a number of autonomic elements, in interaction or not. An autonomic element is an executable software unit that exhibits autonomic properties, *i.e.* the self-* properties introduced previously. To do so, it implements a control loop in order to constantly meet high-level goals set by authorized entities.

Specifically, an autonomic element regularly senses the possible sources of change through sensors, reasons about the current situation and performs adaptations through actuators when and where it is necessary. Any action that an autonomic element executes is done in order to better satisfy its goals given the current situation. At this level of abstraction we are only concerned with the behaviour of an autonomic element as observable from the outside, while ignoring its internal implementation. Indeed, an autonomic element can be rational (or 'smart') or entirely reflex-based; as well as hard-coded or equipped with self-learning capabilities. What an autonomic element does at a given time depends on its goals, on what it perceives and possibly on its knowledge.

An autonomic element can be seen as an independent software module. It is however driven by information provided by administrators, human or not, including goals to be pursued, policies and strategies to be employed. An autonomic element has to report to administrators. It has to provide them with under-

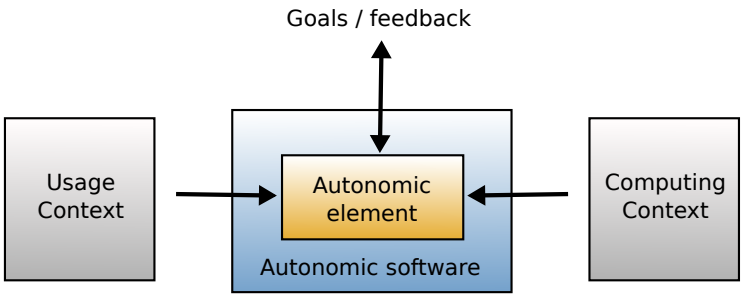


Figure 2.1: Autonomic element [LDJ13]

standable feedback so that they can be aware of its internal situation and degree of success in attaining its goals. Administrators can thus adjust goals or revise strategies when appropriate.

An autonomic element generally relies on a control loop, which monitors the current state of the element and changes it if needed. Specifically, the architecture proposed by IBM makes a clear distinction between the managed resources and a control module called "autonomic manager" (Figure 2.1). Managed resources are the software or hardware entities that are automatically administered in an autonomic element. They can represent, for instance, a Web server, a database, a virtual machine or a server's CPU resource. The autonomic manager is a software module that implements an advanced control loop as introduced here - the MAPE-K loop. The autonomic manager is responsible of the runtime administration of its managed resource and more generally of the autonomic element it belongs to.

Managed resources provide specific interfaces, called control points or touch points, for monitoring and adaptation. Two types of touch points have been defined: sensors and effectors. Sensors provide information about the managed resources. They are materialized by some code that measures a physical or abstract quantity concerning the managed resource and converts it into a signal for the autonomic manager. This could be some information regarding the elements' state or some idea of their current performance. For a web-server for example, such measure could include the response time to client requests, network and disk usage levels, or CPU and memory utilisation. Other examples of such data include system performance characteristics, user context or even server temperature.

Depending on the targeted autonomic properties, different types of data and different forms of presentations can be needed to conduct self-management actions. Determining the appropriate data to be collected and implementing the corresponding sensors is considered today as a difficult activity. It demands to find an appropriate balance between the amount of collected data and the cost of obtaining the data (monitoring has always a cost). Therefore, given the complexity and cost of instrumenting a system, the aim is not to collect just any information that can be obtained about a system but rather to get *appropriate* data that can be used to carry out autonomic management.

Effectors provide interfaces to control/modify the managed resources and, as a consequence, to change their behaviour. For instance, this could represent some modification of a configuration file, the instantiation of new objects, the deletion of a component, the replacement of some outdated elements, and so on. Effectors are also materialized by some code that effects changes and that are provided by the managed elements. The purpose of effectors is to allow the autonomic manager to trigger modifications to the managed elements in a coherent and controlled fashion. If we go back to the web server example, sensors will obtain data about the server load and the number of live connections, while effectors will change the configuration properties of the server.

2.4.2 The MAPE-K loop

The purpose of an autonomic manager is to adapt a set of managed resources at runtime, in response to internal or external changes, in order to achieve predefined goals. An autonomic manager is structured around a collect/decide/act control loop, which may also rely on some shared knowledge.

According to IBM's vision, this reference control loop is composed of a set of tasks executed in a repetitive manner. This loop is known under the MAPE-K acronym, for Monitor, Analyze, Plan, and Execute. K stands for Knowledge, in reference to the knowledge that is needed to conduct the aforementioned tasks. This loop is illustrated in Figure 2.2. This approach relies on the well-known pattern of observation/diagnostic/solution/treatment.

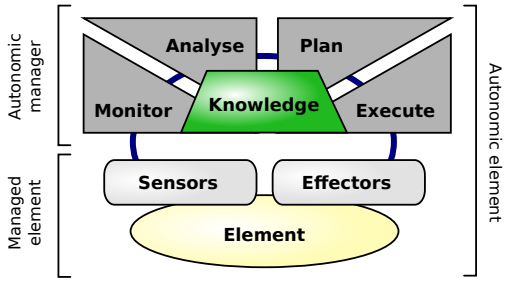


Figure 2.2: MAPE-K loop of Autonomic Computing Element

The MAPE-K logical architecture has profoundly impacted the autonomic computing field, providing a structuring framework to start with when building an autonomic system. It is a modular architecture making sense for practitioners and combining properties like separation-of-concerns and scalability [LDJ13]. The different management activities, defined in rather abstract terms, take care of focused, well-defined and complementary aspects. Standardising the communication interfaces of these activities, as advocated by IBM, would allow the easier integration of various techniques developed by different providers. The architecture also features a certain degree of scalability since management activities can be executed on different machines, assuming that this is correlated with network latency and does not affect reactivity.

For each stage of the MAPE-K loop, researchers have investigated the use of different techniques and tools. Figure 2.3 outlines some of the related works. In the following, we will detail these four stages in depth.

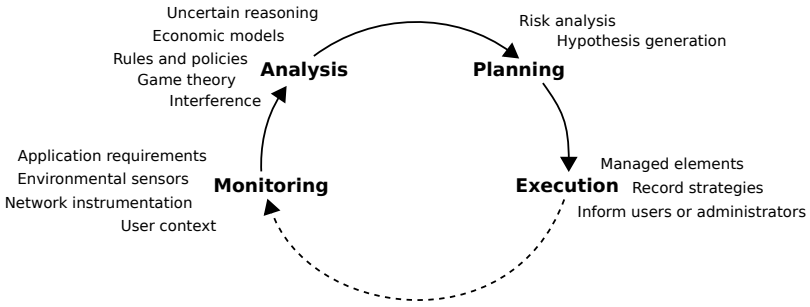


Figure 2.3: Technologies applied to the four stages of the autonomic manager (adapted from [DSNH10])

Once again, Figure 2.3 highlights the fact that autonomic computing has to identify and adapt techniques and technologies from multiple domains. Indeed, this "Monitor, Analyse, Plan, Execute" loop clearly draws inspiration from domains like Control Theory, Artificial Intelligence and Robotics. These domains have in fact common points and are often used in conjunction, for instance when it comes to the design and implementation of automates and autonomous robots.

It is important to note that the MAPE-K loop represents a logical architecture and is not intended to be literally implemented as such in all autonomic systems [LDJ13]. Rather, its purpose is to indicate the main functions that an autonomic manager must support for administering a system and the main interdependencies between these functions – i.e. analysis depending on monitoring, planning on analysis, execution on planning, and all on shared knowledge. It also shows the manner in which the autonomic manager interacts with managed resources - via sensor and effector touchpoints. Various concrete designs and implementations are possible to instantiate this reference architecture.

Indeed, the MAPE-K proposal is not always directly applicable. For instance, having the control flow going through the well-defined interfaces of the four management activities has a performance cost that cannot be always afforded. Thus, in some cases, grouping together some management activities, like analysis and planning for instance, can be required to meet a given deadline. Conversely, having each activity implemented via a single software module is not always feasible. For example, having one autonomic manager in charge of several heterogeneous resources may require multiple monitoring and/or execution modules.

2.4.3 Monitoring

The monitoring part of an Autonomic Manager collects, aggregates, and filters data obtained from the managed element and, possibly, from other relevant entities. Monitoring tasks directly use the Sensor touch points made available by the managed elements and/or by the execution environment. The purpose of monitoring is to collect 'useful' data providing a synthesised view of the execution process and/or context. Such data, once filtered and appropriately formatted, is then analysed by the autonomic manager, which may then undertake correcting actions in response to changes or deviations from the objectives set by administrators.

The monitoring phase involves several monitoring techniques that depend on the kind of information that is needed by the autonomic element to perform its work appropriately. For instance, in the case of resource allocation in a Grid environment for self-optimizing or self-healing purposes, it is necessary to monitor CPU and memory utilization of each active node in the grid. The autonomic manager then analyses this data to detect failures or suboptimal performances and to consequently take appropriate decisions. Sensors are often application specific. Hence, autonomic frameworks generally provide a special-purpose API to implement specific sensors for particular targeted systems.

A well-known issue is that monitoring can be a rather costly process. In fact, there is a trade-off to be made between the data that are needed to understand the state of the system and to perform appropriate actions, and the actual cost of obtaining the data. Much work in the autonomic computing research community has focused precisely on how to decide which subset of the many performance metrics that can be collected from a dynamic environment should be actually obtained from the performance tools available [LDJ13].

Also, data to be collected depend on the goals and on the state of the management process. Goals set by administrators clearly affect the way in which monitoring should be performed. The focus of monitoring can also change as a function of the human administrators' interests. Similarly, intermediary results about the situation of the managed artefacts regarding the goals can influence data to be monitored and the way to collect them.

Further, a more dynamic approach to the monitoring of systems to facilitate autonomicity can be taken. For example, Agarwala et al. [ACMS06] propose QMON, an autonomic monitor that adapts the monitoring frequency and therefore monitoring data volume to minimize the overhead of continuous monitoring while maximizing the utility of the performance data. It could be described as an autonomic monitor for autonomic systems [LDJ13].

There are two kinds of monitoring in autonomic computing systems [HW05]: passive and active monitoring.

Passive monitoring consists in sensing a software or hardware component with a specialized monitoring entity. Most often, targeted systems feature embedded monitoring components. In Linux for instance, the 'top' command returns information about CPU utilization by each process. Another example in the context of virtual machines, clusters, and Grids is the Ganglia Monitoring System [MCC04]. This system relies on a multicast-based listen/announce protocol to monitor the state of targeted systems. To illustrate this, Figure 2.4 presents a live report of Ganglia while monitoring the Wikipedia infrastructure. These information can be exploited by autonomic managers to self-manage the corresponding system.

The purpose of active monitoring is to capture low-level information from the target system components through the use of techniques like code injection or aspect-oriented programming. This is a particularly interesting option to let autonomic managers inject probes to the targeted system when needed and for the specific information they need. An interesting example of such type of monitoring can be found in [MGGL11]. In this work, a set of probes are injected at different levels of a data-mediation chain to obtain relevant monitoring data as needed.

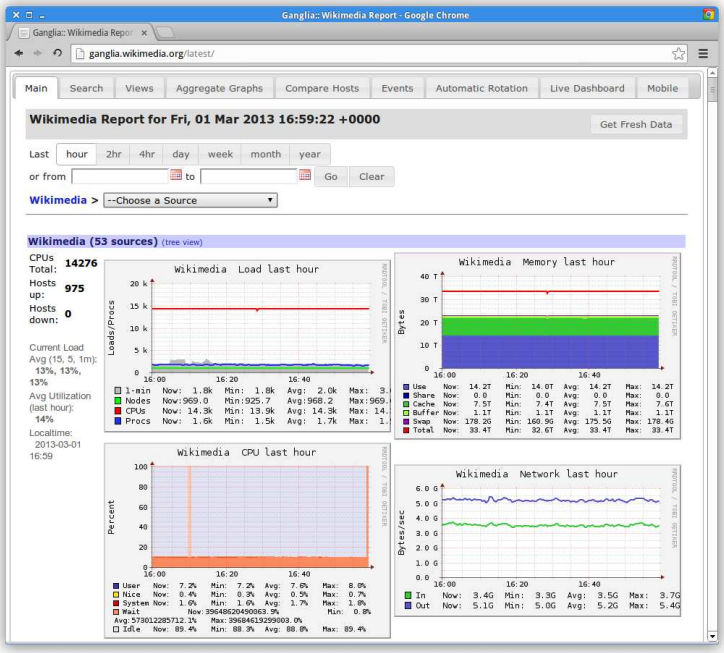


Figure 2.4: Ganglia Monitoring System Report of Wikipedia Servers

2.4.4 Analysis

The purpose of the analysis process is to evaluate the current state of the targeted system and of the execution environment. To do so, it uses the monitored data, models the situation and analyses it to

determine if changes need to be made to the managed system. A wide variety of algorithms and techniques can be employed for detecting misbehaviours or insufficient quality of service, correlating information, anticipating bad situations, identifying problems and defining a better state to be reached. Hence, the analysis phase can define the desirable state(s) that the managed system should be shifted towards. The planning phase - the next one in the loop - is then responsible for preparing a set of actions to be carried out in order to achieve such states.

As said, the techniques available to perform analysis are numerous. They include the construction of models for evaluating a situation, classification approaches to determine when constraints are not met, or learning systems to better apprehend situations in advance. Prediction systems typically monitor trends and foresee if a constraint or goal is to be broken in the near or remote future. This can be implemented with simple regression analysis or with more complex techniques like Hidden Markov Models that represent temporal states of the system and can be used to model the outcomes of a plan.

Often at this phase, an analyser also takes as an input the objectives and/or policies specified by administrators, along with the monitored data and the acquired knowledge. Three types of policies have been largely used by the autonomic computing community to express such administrative goals: event-condition-action (ECA) policies, goal policies and utility functions [KW04]. ECA policies take the general form: "when *event* occurs and *condition* holds then execute *action*". Here, the analyser detects the event from the monitored data, evaluates the condition specified by the user and then plans an action or a set of actions to be executed (which is theoretically part of the planning phase). As an example, such ECA policy may state that "when 95% of Web servers' response time exceeds $2s$ and there are available resources, then increase the number of active Web servers".

While ECA policies can be relatively straightforward to implement, they feature several limitations that can make them unsuitable in certain management contexts. Notably, if the execution context or managed resources undergo dramatic changes that were not predicted when the ECA rules were specified, then this type of management logic cannot adapt itself to the new conditions. Indeed, since the administrative goals are not explicit in the policy definitions the autonomic manager has no way of knowing how to modify its reactions in order to achieve them in the new context. Goal-oriented policies address this limitation by explicitly defining goals and allowing the autonomic manager to infer its decision logic automatically in order to achieve these goals case-by-case. The increased difficulty of implementing the inference process is the main downside of the goal-oriented approach.

Another important difficulty in analysing monitoring events comes from the conflicts that can occur between polices or objectives specified for an autonomic element. When several ECA policies are triggered by the same event, their actions can be conflicting and the resulting state of the managed element incoherent. Administrators must take great care (or use special-purpose tools) for preventing such conflicting policies from being defined. When a goal-oriented perspective is adopted, administrators may want an autonomic element to pursue several goals at once; these goals may be conflicting (e.g. a Web server's performance and security objectives). While goal-oriented policies are binary – the goal is either met or not met – utility functions can express various degrees of success with respect to specified goal(s). More precisely, utility functions are defined so as to be able to quantify the level of desirability of (monitored) data and its implications. Hence, the desirability of a given state can be qualified with respect to several targeted goals. The utility measure is expressed as a function, which takes as input a number of state parameters and provides as output the state's desirability rating. Utility functions can be employed both during the analysis phase – to evaluate the current system state – and the planning phase – to evaluate the predicted outcomes of various actions and select the optimal action to execute. The major difficulty with utility functions is that they can be hard to define, as all relevant aspects must be quantified.

2.4.5 Planning and Execution

Planning and execution are complementary activities. Planning focuses on high-level actions to be undertaken while execution is much more concrete in the sense that it deals with the implementation of the plans and the direct interaction with the artefacts under execution. Separating planning and execution has been much investigated in Artificial Intelligence in order to deal with complex environments that are difficult to observe and to predict.

The purpose of the planning activity is to come up with a set of actions to be carried out on the managed artefacts in order to achieve the administrators' goals. The planner should not consider the implementation details of the actions but rather remain at a certain level of abstraction in order not to be influenced by low level evolutions of the managed elements. The planner relies on a set of actions that can be performed on the managed system. These actions are strongly related to aspects of the managed system, and are executed by specific effectors.

There are two different ways to address planning in autonomic computing systems: domain-specific or generic. In the first approach, the expertise of system administrators is incorporated in the planner as a set of policies and rules. This is the case of ECA rules where actions are written by administrators based on their experience in the domain. In the second approach, the idea is to formally express the problem, or the state to be reached. The planner relies on generic algorithms to determine the sequence of actions to be taken depending on the pre-conditions and effects of each action on the state of the targeted system. In several related works, the action plan is not directly executed on the target system, but in an intermediate architectural model that reflects the managed system. This architectural model is used to formalize the current state of the system; that is, its structure, its behaviour, and even its requirements and objectives.

The purpose of the execution phase is to schedule and control the execution of the action plans produced by the planner. Clearly, execution is a much more concrete activity than planning in the sense that it directly interacts with the managed artefacts. A plan, for instance, could specify a set of parameters to be changed, with no ordering constraints. The execution activity, then, has to determine how and when the parameters have to be changed (e.g. to ensure the integrity and the coherence of the resulting system state). It is a matter of timeliness and synchronization where a number of functional and non functional dependencies have to be considered. Usually, a parameter can be changed only if some conditions hold. When several parameters have to be modified, ordering constraints have to be respected.

Concrete actions are performed through effectors, which are system-specific and have access to the managed system elements. Such action can consist in simply reconfiguring some properties, in changing the very structure of a component-based application, or in modifying the behaviour of some components.

2.5 Engineering autonomic systems

As previously stated, the aforementioned reference architecture for autonomic computing systems - the MAPE-K loop - is mainly inspired from control theory (see section 3.1 of chapter 3). The MAPE-K architectural blueprint provides a valuable conceptual architecture to implement autonomic managers. It remains however at a high level of abstraction and, obviously, concrete implementations may differ. Some functions can be modified or even omitted, others can be added. Indeed, different implementation variants of the MAPE-K loop have been proposed in the literature.

In this section, we explore significant propositions regarding the MAPE-K implementation. Our presentation will focus on the way in which autonomic managers are structured and architected to fulfil the desired goals and to reach the expected quality attributes. Indeed, making complex computing systems self-managed involves more internal complexity. It is likely that autonomic elements will have complex

lifecycles, continuously carrying out multiple threads of activity, and constantly sensing and responding to their execution environment. It is clear that engineering such systems is an essential issue.

In the next sub-section, we outline some important quality attributes that autonomic systems should have. These attributes will be used as comparison criteria for the autonomic frameworks studied later on.

2.5.1 Quality attributes

Along with the aforementioned autonomic properties, also called self-star (self-*) properties [BJM+05b], an autonomic computing systems should also provide, like any other modern software system, certain quality properties [LML05] such as adaptability, dynamicity, scalability, robustness, or interoperability.

Hereafter, we define some quality attributes that we believe to be essential for autonomic systems and that are in the scope of the thesis:

- **Adaptability**. In order to cope with changing user requirements and with market and runtime conditions, autonomic computing systems should remain flexible after their deployment. Note that adaptability is equivalent to the term *flexibility* in the IEEE Standard Glossary for Software Engineering Terminology [EE90], which defines it as "*the ease with which a system or component can be modified for use in applications or environments other than those for which it was specifically designed*". The core concept is the general ability to change a system's observable behaviour, structure or realization. Adaptation may affect a system's functionalities, algorithms or parameters, as well as its overall structure.
- **Dynamicity**. Dynamicity is the ability to add, remove or exchange some code under the form of components, modules or services during *runtime*. In contrast to adaptability, this quality only constitutes a technical facility of change during system execution [Esc08]. Dynamicity deals with concerns like starting and stopping a functionality, preserving states during a functionality's update process, and so on. There are close but not dependable relationships between dynamicity and adaptability. Some researchers refer to dynamicity via the terms *Dynamic Adaptation* [FC09] or *Adaptiveness* [AAB+07].
- **Scalability**. Scalability refers to the ability to cope smoothly with an increase in the workload [DRW06]. The specific type of workload will depend on each system concerned. In practice, scalability can be declined in different ways. For instance, it can refer to the ability of a server system to always respond satisfactorily despite an increasing number of requests. It can also refer to the ability of a network to support the addition of nodes. In an autonomic computing context, it can refer to the ability of an autonomic management solution to deal with an increasing number of managed elements or with a high frequency of changes.
- **Robustness**. Autonomic systems cannot guarantee to operate so as to avoid all system malfunctions [Cap10]. Consequently, the robustness quality of autonomic systems is an important feature. It is defined as "*the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions*" [EE90]. This is related to other properties such as fault-tolerance and self-healing. Autonomic systems must be robust as the self-healing function requires them to detect and correct problems such as errors or communication losses.
- **Interoperability**. Interoperability is defined as "*the ability of two or more systems or components to exchange information and to use the information that has been exchanged*" [EE90]. In the context of autonomic computing systems, this refers to autonomic management components that coexist and interact - either directly or indirectly - within the same autonomic application or system [APS11]. A multi-manager presence with potentially different objectives may lead to conflicts, where the actions of one manager can have an undesirable impact on the management functions of the other managers.

Autonomic frameworks should provide mechanisms to deal with such problems and hence increase the level of interoperability between the different autonomic managers - e.g., [FDMD13].

- **Reuse**. Software reuse is the process of creating software systems from existing software rather than building them from scratch [Sam97]. As the different components of an autonomic computing system are often highly difficult and costly to build, there is a significant demand for reusing autonomic components across different autonomic systems.

2.5.2 Autonomic Frameworks

There is a lot of work around Autonomic Computing Systems, tackling different problems, challenges, techniques and applications [HM08]. In this section, we focus on the software engineering point of view, by studying and outlining some of the frameworks that have been proposed to help build autonomic computing systems. To conclude this section, we present a comparative study highlighting the quality attributes of each studied framework.

Ceylon

Ceylon [DML08, MDL10, Mau10] is a project developed at Grenoble University. It proposes to build autonomic managers by integrating specialized administrative tasks, where each task implements a specific management aspect, such as monitoring a certain feature of the targeted system, detecting specific problems, planning a specific solution or modifying a managed resource in a certain way. Task integration is performed dynamically in order to produce management chains that are adapted to changing conditions and requirements. Tasks can be known at deployment time and/or discovered and brought into the system at runtime. Depending on the execution context and the state of the managed application, a high level *Tasks Manager* activates or deactivates specific tasks and forms a chain of tasks for dealing with the observed problem, in accordance with higher-level management strategies specified by the administrator (see Figure 2.5). This approach enables the opportunistic composition of simple tasks into more complicated management processes that can deal with complex and unpredictable situations. The management tasks are implemented as services, which communicate asynchronously through the exchange of topic-based events.

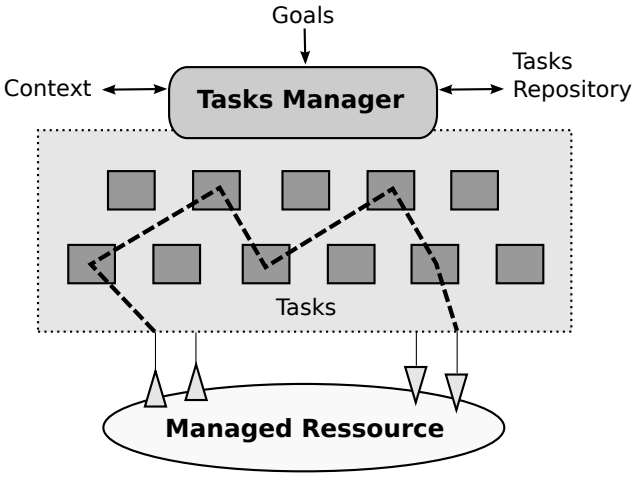


Figure 2.5: Ceylon framework

Despite the advantages of using a dynamic architecture to choose and switch between different management tasks and the possibility to manage multiple aspects at the same time, we believe that this framework lacks several interesting features. Most importantly, relying on a centralised Task Manager will limit, sooner or later, the scalability of a Ceylon-based management solution. Indeed, large-scale applications featuring a significant number of distributed, heterogeneous and highly-dynamic managed elements raise complex administrative problems, which are difficult to handle via a central entity. In such cases, specialized tasks can be implemented to retrieve data from distributed resources, execute actions on heterogeneous elements, or be dynamically extended to deal with new components or situations. Yet, Ceylon's centralization with respect to the task integration process may lead to overloading situations and constitutes a single point of failure.

This work is part of a bigger trend to use service-oriented computing to design loosely-coupled autonomic managers with dynamically interchangeable elements. Such efforts aim to standardize their interfaces to facilitate the reuse of multiple implementations. Miller [Mil05] showed the advantages of standardizing the interfaces of autonomic management elements; this is motivated by the arrival of new standards for the distributed management of resources, like the Web Services Distributed Management (WSDM) [Sed04].

Rainbow

Rainbow [GCH+04] is a project developed at the Carnegie Mellon University by David Garlan and his colleagues. They investigated the use of architectural models at runtime to help the dynamic adaptation of software systems (see section 3.4 of chapter 3). It is rather an overall approach for engineering self-adaptive systems, comprising a framework, an adaptation language and an adaptation engineering process. The framework includes a closed-loop control that monitors the system and reasons about appropriate adaptations using the architecture model.

Figure 2.6 depicts a system under Rainbow's control. In particular, it highlights the two layers of the Rainbow architecture (*system* and *architecture* layer) and the underlying framework elements. At the system layer, a set of *Probes* monitor the running system, and update properties of the system's architectural model. The architectural model is managed by a *Model Manage*r, which belongs to the architecture layer. An *Architecture Evaluator* checks constantly that the system is operating correctly (within acceptable bounds), as defined by architectural constraints specified by administrators. If something goes wrong, an *Adaptation Manager* analyses the architectural model, which reflects the current monitored system, and selects the best strategy to be undertaken. A *Strategy Executor* performs the selected strategy via system-level *Effectors*. The possible adaptation actions depend on the underlying managed infrastructure; if the targeted system is component-based software with the capability of reconfiguring the components and changing the connectors at runtime, then specific Effectors can be implemented to execute such actions. A specific Strategy Executor is also implemented to choose such adaptation strategy while reasoning about the action to be undertaken upon constraints violation on the architecture model.

Most of the Rainbow framework's components can be reused across different systems and use cases. For instance, engineers can develop probes and effectors for a specific system and then reuse them for other compatible systems. Most importantly, the Architecture Evaluators and Strategy Executors can be reused across several projects, since their reasoning is based solely on the runtime architectural model and not on the underlying managed infrastructure.

Several Rainbow sub-projects were also developed, including *DiscoTect* [YGS+04] - which aims to provide architectural views by observing a running system - and *RAIDE* [CGS09] - which enables adaptation engineers to customize the Rainbow framework, simulate adaptation behaviour and deploy Rainbow runtime components.

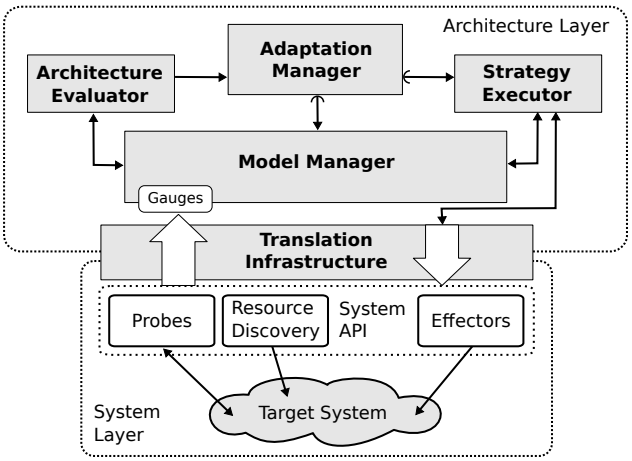


Figure 2.6: Rainbow framework

As before, an important limitation of the Rainbow framework can occur from its centralized approach to system management. This can raise scalability issues and introduces a single point of failure. If the centralized element fails, the entire system may be stopped. Some lines of work have been explored in [wCcHG+04] to coordinate the operation of multiple specialized managers on a single system. However, this approach simply allows to decompose the administration function into several entities rather than to manage the system in a decentralized manner. Moreover, the architectural model that is used as a reference for the adaptation process is highly precise (or 'concrete') in the Rainbow approach. This means that the adaptation process must ensure that the runtime architecture of the managed system is identical to this reference architectural model. While this is highly useful for attaining self-repair functions, it also limits the space of acceptable system variations or adaptations.

We notice finally that the Rainbow approach is among other research efforts that use dynamic, architecture-based adaptation for the self-adaptation purpose [BCDW04].

Unity

Unity [CSWW04, TCW+04] is a project developed at the IBM Thomas J. Watson research centre. The project's objective is to enable the autonomic management of distributed software systems. Unity has focused primarily on the autonomic management of applications running on clusters and Grids. A specific objective of Unity is to provide an architecture for enabling the autonomic allocation of resources, such as servers, to different applications, taking into account the policies specified by administrators.

A basic assumption made in this work is that all resources, both physical and software, are represented in the form of autonomic elements. Each element is responsible for its internal state, as well as for its dependencies with other autonomic elements (agents) to achieve their goals. Technically, the autonomic elements are developed in the Java programming language using a set of autonomic tools [JLHNY04] provided by IBM. Communication between the different elements is achieved by means of Web service calls.

Figure 2.7 shows the different elements that are part of the Unity framework. The different autonomic elements of Unity do not know each other in advance, but they can be discovered during execution (using a shared directory). Unity proposes the use of a repository of policies through which the user is able to control the system. The user can add or remove policies, and applications connect to this repository to

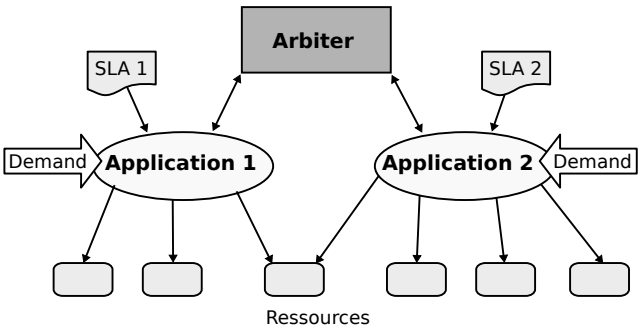


Figure 2.7: Unity framework

discover the policies that govern their operations. Policies are specified in the form of *objective functions* (or *utility functions*). To determine how many servers should be allocated to each application, the Arbiter executes an algorithm to maximize all objective functions for all applications, and hence determines the optimal allocation of servers for applications. Hence, the Arbiter aims to optimise the utility of the cluster or grid system overall.

Unity solves the problem of the autonomic resource allocation in the presence of multiple applications with different needs in a cluster or a grid. This project has a number of advantages. First, the systematic use of autonomic elements to represent hardware and software resources allows standardization of managed elements. Second, the project proposes the use of policies expressed via objective functions. This type of policy has the advantage of allowing the selection of an optimal system configuration, compared to other goal-oriented policies that cannot choose between several solutions satisfying the requirements, or make a compromise if no solution can meet all the requirements. On the other hand, the project provides a repository of policies that is the only user interface. This technique simplifies system management and allows the user to dynamically change the policies associated with the self-managed system.

Unity also features a few limitations. The first one comes from the extreme specialization of the Unity project where the principal objective is to provide a system for self-optimising resource allocation of several applications. This framework cannot be used for implementing other self-* capabilities such as self-repair or self-protection. Also, the framework imposes that each managed application is able to provide a rather detailed utility or objective function, which can be difficult to specify. Finally, as with the previous frameworks, Unity's control architecture is centralised, since all actions are taken by the Arbiter for resource allocations.

AutoMate

AutoMate [PLL$^+$06] is a project developed at Rutgers University. It aims to investigate key technologies to enable the development of autonomic applications that manage complexity, dynamism, and uncertainty associated with runtime grid environments. It includes sub-projects like *Accord* [LPM06] - the autonomic programming model; *Rudder* [LP04] - a rule-based multi-agent infrastructure; and *Meteor* [JQSP08] - a content-based interaction middleware infrastructure.

Conventional paradigms based on passive components and static compositions are becoming insufficient to deal with the scale, complexity, heterogeneity and dynamism of modern computing systems. The research teams developing AutoMate have considered alternative approaches. They used a component-based programming framework, called Accord [LPH04, LPM06]. The main advantage of using such component-based infrastructure is the clear separation between functional and non-functional aspects, in

particular, the separation of composition rules and autonomic management from functional code of the application. An AutoMate component includes some application-specific logic and an attached agent that adapts its function and interaction (see Figure 2.8). The agent has a set of policies which are expressed as *Event-Condition-Action* rules (ECA). These rules can be dynamically injected or withdrawn from the component. They allow the management of the autonomic component, but also its interaction with other components. The inputs/outputs of an autonomic element's agent are illustrated in Figure 2.8. An AutoMate agent corresponds to an autonomic manager as defined in the autonomic computing reference architecture proposed by IBM. Monitoring information consist of data representing the state of the component, its execution environment context, and a set of composition or management rules. As output, the agent can act on local or remote effectors, and it can use the underlying peer-to-peer infrastructure to adapt its composition with other components.

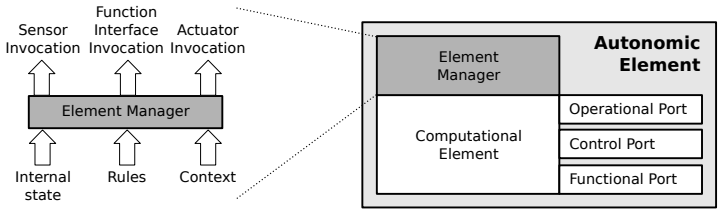


Figure 2.8: AutoMate Autonomic Element [PLL+06]

AutoMate proposes to manage applications by attaching a handler to each component of the autonomic system. In addition, to simplify the expression of the rules, AutoMate proposes a workflow to specify the different application rules. A Composition Manager is then responsible for interpreting the rules specified in the application level and injecting the results into the various components (as illustrated in Figure 2.9). This system allows expressing autonomic management rules in one place, while managing the application in distributed manner.

Compared to the IBM's reference architecture, AutoMate makes a set of choices, such as a component model for the implementation of applications on which the autonomic reasoning must take place, and its associated runtime reconfigurations. AutoMate adds the notion of global policy rules expressed in the application level, which is then split and interpreted by each autonomic element. However, AutoMate limits the policy specification to only ECA rules. Indeed, this type of policy is limited when it comes to the expression of complex behaviours. In addition, the management architecture proposed by AutoMate is fully decentralised, which can cause some problems for managing conflicts among decisions taken by different managers.

Auto-Home

Auto-Home [Bou08, BDLM11] is a project initiated at the University of Grenoble. It provides solutions to assist designers to develop autonomic applications based on hierarchical service-oriented architectures. It focuses on pervasive home systems as its application domain.

At the top level, the main manager has a global view of the objectives to be achieved. It delegates some of its responsibilities to lower-level managers with some fixed sub-objectives. In this way, the goals become more specific and correspond to detailed objectives making sense at the lowest levels. Hence, the leaves of the tree have no awareness of the global objectives. On the other hand, when a lower manager cannot find a solution to a given problem, it can forward the problem to its superior, which can deal with more abstract information and with a more global view.

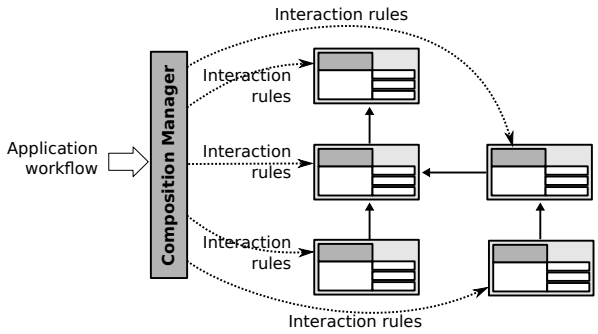


Figure 2.9: AutoMate Composition Manager

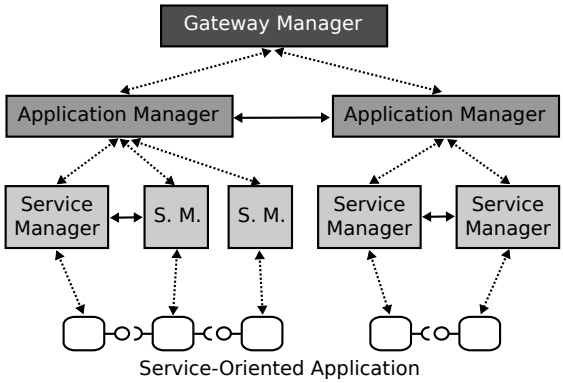


Figure 2.10: Auto-Home Hierarchical Architecture

Figure 2.10 illustrates the three hierarchical levels of Auto-Home self-management framework in one (centralized) execution platform:

- The **Gateway Manager** has the overall view of the managed home system. Its role is to manage the platform resources and their allocation to different applications. It must ensure the proper functioning of the platform and its applications. It can make the decision to stop and remove an application in order to allow other applications to work properly. For instance, an alarm is more important than the displaying of weather information. This manager has a global view of all the activities of the platform.
- An **Application Manager** administers one application in the home system. It aims to maintain the proper functioning of the application by ensuring the proper execution of its underlying services. For instance, it can do so by choosing the most appropriate service providers. The Application Manager carries out application policies specified by the administrator. It should, for instance, be able to limit the resources occupied by its managed application if the Gateway Manager requests this behaviour. The Application Manager has a global view of the application, allowing it to take consistent decisions at this level.
- A **Service Manager** administers a single application service. Hence, it has a limited perception of its environment and must follow policies set by its Application Manager.

The goal of the Auto-Home approach is not to have an emerging behaviour from the bottom management levels. It is rather to apply control decisions and ensure user policies specified from the top level. Each

manager tries to optimize the resources under its control. Objectives of different managers can be opposed, for example when two applications want to control the same resource. Auto-Home's hierarchy of management and control allows solving this type of conflicts.

One of the main drawbacks of this approach is that messages exchanged between managers are not clearly specified. This kind of information is frequently defined differently for each situation in an *ad-hoc* manner. Moreover, the internal architecture of managers is not specified and appears to be monolithic, which can be a major handicap for maintainability and reusability.

Jade

Jade [SBDP08, DPBB+08] is a project developed by the Sardes Team of INRIA institute in France. Its main objective is to enable the autonomic management of legacy applications. Jade proposes to encapsulate them into standardised components with unified administration interfaces, and then to apply reconfiguration plans onto it.

This project was originally designed for the self-management of JEE[2] application servers. Since then, it has been extended to other types of distributed applications, in particular message-oriented middleware. The objective is to provide a tool which, from the description of the application's architecture will enable self-healing, self-configuration, and self-optimization of the application. In this system, there is a set of managers that handle each aspect of the application. In addition, Jade automatically maintains a knowledge base with the current architecture of the managed system. Administration policies are implemented specifically to access and use the built-in architectural view during runtime. Developers should also implement specific touchpoints and the autonomic manager's adaptation code to detect anomalies and apply changes. The possible adaptation operations are predefined. They can be accessed via the administration programming interface exposed by the standardised components encapsulating the legacy system. Figure 2.11 illustrates the encapsulation principal of a Jade system.

Jade autonomic framework was implemented based on the Fractal component model [BCL+06]. Fractal is a general component model that is intended to implement, deploy, and manage (i.e. monitor, control, and dynamically configure) complex software systems, including in particular operating systems and middleware. Jade uses Fractal components as an abstraction layer for the legacy software or hardware elements to be managed. It benefits from the introspection and re-configuration capabilities of Fractal components to allow programmers to code control features for managed components represented in the abstraction layer.

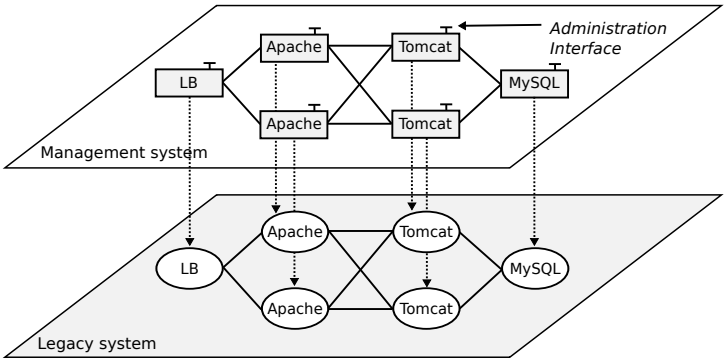


Figure 2.11: Encapsulating legacy systems with Jade

[2]Java Enterprise Edition

Component-based administration as proposed by Jade allows:

- Managing heterogeneous elements through a uniform programming interface instead of using specific interfaces for each element (usually configuration files and scripts).
- Managing complex environments from different points of view. Indeed, using Fractal components and composites allows encapsulating complex systems at various granularities or abstraction levels (e.g. network infrastructures, operating systems or middleware).

After defining the management layer as consisting of components and composites (also known as Managed Elements), exposing well-defined administration interfaces, Jade allows adding administration and control programs (i.e., autonomic managers) that analyse and act on the abstraction layer to ensure self-management of the encapsulated legacy system (see Figure 2.12). Jade provides a set of common services and two main autonomic managers:

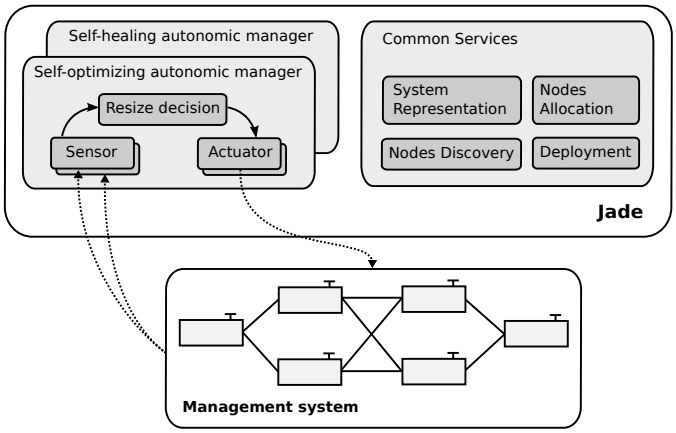


Figure 2.12: Jade autonomic framework

- *Self-healing manager* [BBH+05]: a set of probes are deployed to monitor the execution nodes of a JEE application. The self-healing manager monitors these nodes periodically and when detecting a failure it allocates and activates a replacement machine; redeploys the necessary management elements; and ensures consistent repair of the broken part.
- *Self-optimizing manager* [BPHT06]: this manager handles the overload of a replicated JEE architecture. It uses a detection mechanism to determine when response time becomes unacceptably large. When this is the case, the self-optimising manager uses available machines to create a new instance of the system.

The main limitations of Jade correspond to the high level expertise needed to build autonomic managers. Indeed, management policies are hard-coded. Second, the centralized management architecture proposed features the main limitations as the previous frameworks discussed – i.e. scalability and robustness limitations. Finally, Jade does not provide a mechanism to manage the coordination between the different managers – self-healing and self-optimisation.

2.5.3 Comparison

In our comparative study, we are interested in the three complementary points of view:

1. Control logic placement and its possible organization;

2. Application spectrum (i.e., generic or specific), supported self-star properties, and goal specification approach;
3. Supported Quality Attributes.

Table 2.2 depicts the results of our comparison regarding the first point of view. In this case, we consider two aspects of the control logic: (1) how the control code is placed, and (2) how the control components (i.e., autonomic managers) are organized (i.e. architected) to fulfil the management objectives. Control logic is considered "internal" if it is injected directly into the managed system, that is, if it is part of it. The term "external control" is used when a separate system monitors the managed system and acts upon it. Each of the two possibilities has advantages and drawbacks. For internal code placement, developers have a deeper control of the managed elements. Techniques like reflection or container-based control are used in this case. Among the studied frameworks, AutoMate has a dedicated programming model (Accord) that explicitly defines autonomic components with functional and control code. Hence, the control logic is part of the component and is handled by an agent that interacts with other agents to fulfil global objectives.All the other frameworks exemplified follow an external code placement approach. This is mainly inspired from control theory (detailed in the next chapter, section 3.1). The main advantage of such an approach is the separation between target systems and control systems, so that the same control components can be used to control several targeted systems (often legacy).

Table 2.2: Comparison of control placement and organization

	Control Placement		**Control Organization**		
	Internal	**External**	**Centralized**	**Hierarchical**	**Decentralized**
Ceylon		x	x		
Rainbow		x	x		
Unity		x	x		
AutoMate	x				x
Auto-Home		x		x	
Jade		x	x		

Next, table 2.2, also indicates how the managers are organized. This is independent from the distribution of code among several physical machines. However, it can provide a first insight into the possible distribution of control over a distributed network. This, in turn, can provide further insight with respect to certain quality attributes such as scalability and robustness. In all cases, the centralized or monolithic control cannot be separated into disparate execution nodes, despite the fact that the target system can be distributed across separate machines. That is, the control decisions are made in a single place. This has the advantage of having better control capabilities as all needed information is available in one place at any one time, so that the autonomic manager can take a well-informed decision. Having a single control point also avoids situations where conflicting decisions are taken concurrently at different network locations. However, centralised organizations feature important limitations in terms of scalability and robustness.

Hierarchical control can be adopted to attain management objectives in large-scale distributed systems. The principal idea is to use the divide-and-conquer strategy for splitting the objectives into different levels. Here, low-level controllers manage parts of the large-scale system, while high-level controllers provide means of coordination for the lower-level controllers. The role of the highest-level controller is to adjust the sub-objectives of lower-level controllers in order to attain system-wide objectives.

In contrast to centralised or hierarchical control, approaches that are completely decentralized lack a unique control point altogether. This means that there is no central entity that controls or supervises all

the others, including controllers or managed elements. Typically, each decentralised controller only uses information available from its local environment, from neighbouring controllers, or from other related controllers, in order to take adequate decisions and actions. The system-wide objectives are achieved as a result of such local actions.

Table 2.3: Comparison of specific/generic solutions and of self-* properties

	Specific	**Generic**	**Self-* properties**	**Goals specification**
Ceylon		x	self-*	Code and configurations
Rainbow		x	self-*	Code / Architectural model-based
Unity	Grids		self-allocation of resources	Utility function
AutoMate	Grids		self-healing	ECA
Auto-Home	Home Automation		self-healing & self-configuration	Code
Jade	Legacy		self-healing & self-optimising	Code

Table 2.3 categorises the studied frameworks in terms of their applicability, supported self-* properties and the way in which their goals are specified by administrators. Rainbow and Ceylon are considered generic frameworks as they can be used to manage any kind of targeted systems. These frameworks provide mechanisms and APIs to implement specific wrappers of the managed systems. In Rainbow, this is done by implementing specific probes and effectors for each targeted systems, along with other extensions in the architecture layer. In Ceylon, this is done by implementing specific management tasks. The other frameworks are specific for particular domains. Unity is mainly used for autonomic resources allocation. AutoMate is specialized for Grid environments. Jade has mainly been used for legacy systems, yet it can be considered as generic since various targeted systems can be wrapped and managed as Fractal components. Finally, Auto-Home aims to control home automation applications.

Finally, table 2.4 summarizes how each of the reviewed autonomic frameworks supports the quality attributes described here before (see 2.5.1). We have used the following notation:

- "o" means the framework does not explicitly exhibit the desired quality attribute; however it does not avoid its fulfilment in principle.
- "+" means the framework somehow supports the quality attribute
- "++" expresses that the corresponding quality is highly supported by the framework.

Table 2.4: Comparison of Quality Attributes

	Adaptability	**Dynamicity**	**Scalability**	**Robustness**	**Intero-perability**	**Reuse**
Ceylon	++	++		o	+	++
Rainbow	++	o		o	o	+
Unity	+	+	+		+	
AutoMate	+		+	+	++	+
Auto-Home	++	++	+	+	+	+
Jade	+			+		+

2.6 Summary

Autonomic computing is an ambitious initiative addressing the runtime management of complex software systems. Its main purpose is to drastically reduce the effort required for system administration and consequently diminish maintenance and evolution costs. The idea is to achieve this by introducing well-coordinated autonomic managers capable of replacing the human experts for intervening rapidly and effectively to repair faults, optimize resource utilization, configure new resources, or secure the system against attacks. The inherent complexity of modern software systems will be consequently reflected in the autonomic management systems that control them. IBM has proposed a reference architecture to organize such self-management systems. This conceptual architecture has been implemented in different ways by several research teams, targeting different quality attributes and different domains. In this chapter, we have reviewed some of the proposed frameworks and we have established a way to compare them.

In the following chapter, we will see in more depth the approaches related to autonomic computing and their associated quality attributes. Several techniques used in our thesis proposal (introduced in chapter 4) are inspired from these related domains, and merged together to form our approach for self-managing complex modern software systems.

Chapter 3

Approaches to Autonomic Computing

Contents

3.1 Control Theory . . . 33
3.1.1 Types of Control Loops . . . 34
3.1.2 Related work to Autonomic Computing . . . 35
3.1.3 Summary . . . 37
3.2 Agent-based systems . . . 37
3.2.1 Structural organisation of multi-agent systems . . . 38
3.2.2 Communication principles . . . 39
3.2.3 Adaptive multi-agent systems . . . 40
3.2.4 Research work related to Autonomic Computing . . . 41
3.2.5 Summary . . . 42
3.3 Nature-inspired approaches . . . 43
3.3.1 Self-organisation and emergence . . . 43
3.3.2 Functional structure and pattern formation . . . 45
3.3.3 Work related to Autonomic Computing . . . 45
3.4 Architecture-based approaches . . . 46
3.4.1 Software Architecture . . . 47
3.4.2 Adaptation concerns . . . 48
3.4.3 Enabling Techniques . . . 48
3.4.4 Summary of architecture-based approaches . . . 52
3.5 Discussion and chapter summary . . . 53

To realize the Autonomic Computing vision and meet its challenges, researchers have investigated the use of techniques, methods and theories from other disciplines and fields to cover different aspects of Autonomic Computing Systems. In this section, we detail some of the significant domains from which we have adopted concepts and techniques to develop our approach to self-managing software systems.

3.1 Control Theory

Control Theory/Engineering is concerned with the management of *dynamic systems* [MT89] that constantly interact with their environment and change their behaviour over time. The output of such system

does not only depend on its inputs, but also on the system state, which changes constantly in response to environmental evolutions. In the software domain, dynamic systems are often called *adaptable software systems* [MSKC04]. Because of the constantly changing state and behaviour of dynamic systems, engineers have to find solutions to ensure the system's important properties and objectives, such as *robustness*, *regulation*, *optimization*, *performance* and *stability*. To achieve these objectives, the control-based paradigm considers two types of system inputs: *control inputs*, provided by a Controller, and *disturbances*, which affect system behaviour in an unpredictable manner [KBE99]. The two systems, that is, the controller and the target system, are often coupled by sensors and actuators, and possibly by some transformation operations. The whole forms a *control-loop*.

Controllers are developed for some intended purpose or control objectives. The common objectives include:

- **Regulation**: ensure that the measured outputs of the target system stay within a given range of values - a viability zone. For example, the utilization of a cluster of web servers is regulated so that one web server never exceeds 70% of its capability. In case a server exceeds this limit, the controller, which measures the server's CPU utilization, activates other available servers in order to absorb some tasks from the overloaded server.
- **Disturbance rejection**: ensure that disturbances acting on the system do not significantly affect the measured output; the output should remain within the viability zone. Considering the previous example, when a long running task (like a backup or a virus scan) is executing on the web server, the overall utilization of the system if maintained at 70%.
- **Optimization**: obtain the best possible values of the measured outputs and apply control values to the target system so as to get the system to function in an optimal manner. In the cluster example, the controller aims to switch-on the minimum number of web servers while ensuring that each server's load is under the 70% threshold.

In addition to such business-specific objectives, most control systems must also feature several generic properties, such as:

- **Stability**: ensure that system's state variables and outputs do not diverge or permanently oscillate when receiving bounded control inputs. Ideally, under constant inputs, the system reaches a stable equilibrium within a bounded period. In the exemplified cluster above, the controller should avoid having a server switched on and off repeatedly within short intervals, as could be the case if reacting too finely to a CPU load that is close to the threshold.
- **Controllability**: ensure that the system can be forced into desirable states by applying appropriate control inputs.
- **Observability**: ensure that the system's state can be observed through measurements of its outputs. If a system state is not observable then the controller will most likely not be able to control the system so as to force it into that state.

3.1.1 Types of Control Loops

Control-loops can be of different types: *open* loops, *closed feedback* loops or *closed feed-forward* loops. In open loops (e.g., automatic toasters and alarm clocks) the controller sets the control inputs for the target system without knowing the system's direct outputs. The controller can however receive information indirectly from the environment or from other systems. In contrast, in feedback control-loops (e.g., thermostats or automotive cruise-control systems) the controller adjusts the control inputs for the target system so as to get the system's measured output to be equal, or as close as possible, to the desired values (i.e.,

reference input). Hence, a feedback process determines how the control will be undertaken at any time depending on the current state of the system, as indicated by its outputs, and on the reference input. The feedback control system acts in a *reactive* manner because it waits until a disturbance affects the outputs of the target system before taking the necessary decisions and actions. The process executes as follows (see Figure 3.1):

1. A deviation of objective, or control error, is determined from the sensed system outputs and their comparison to the reference input;
2. The controller "calculates" control actions;
3. The deviation is eliminated by applying the control actions to the target system's control inputs.

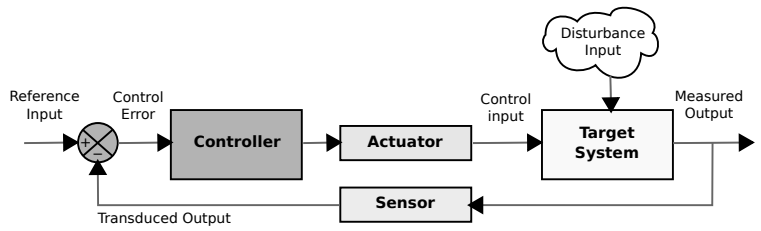


Figure 3.1: Feedback control system

In contrast to feedback control loops, feed-forward control measures disturbances and adjusts the system inputs before the disturbances can affect the system outputs. It acts in a *proactive* manner, as follows (Figure 3.2):

1. A disturbance is determined;
2. A potential future deviation of objective is "calculated", or predicted;
3. A control action is applied to compensate for the predicted deviations.

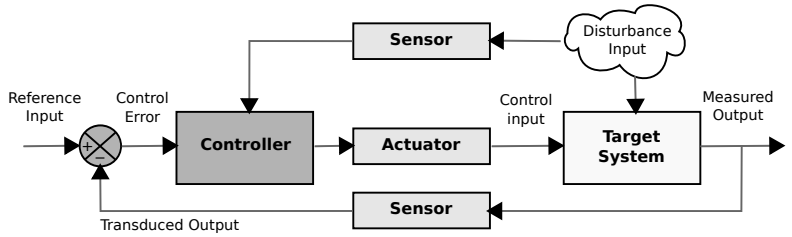


Figure 3.2: Feedback and feed-forward control system

Feed-forward control loops can significantly improve performance compared to feedback control alone, by pre-emptively reducing the future effects of current disturbances (Figure 3.2). However, accurate measurements of disturbances in software systems are hard to acquire [PCHW12].

3.1.2 Related work to Autonomic Computing

Regardless of the obvious analogy between software systems and control systems, the basic paradigm of control has not found its place as a first-class concept in software engineering [KBE99]. In fact, unlike physical and mechanical systems, which are generally dynamic, software systems are often composed of

static modules with off-line configurations. Changes during execution are often not possible, and the restart of the software system is the only way to apply adaptations and new configurations. However, since 2005, advances in adaptive middleware solutions [LRS97, ST09] (e.g., web servers, application servers and business process engines), virtual machines [AFG+04, KKH+09], large-scale Clouds [LBCP09] and Green Computing environments have led to an increasing use of control engineering to self-manage these types of software systems [PCHW12]. The most notable management objectives are performance attributes such as response time, throughput, processor and power utilization. The reason for this could be that the performance attributes like response time are what the user perceives from the system. They often correspond to one of the attributes specified in the service-level agreement (SLA) as part of a service contract. Also, processor utilization is becoming an important attribute because of the increasing demand and cost of power in data centres.

In [HLM+09], a control loop is used to dynamically consolidate the resource consumption in a cluster of computing nodes. Their tool, called *Entropy*, allows administrators to implement scheduling policies through cluster-wide context switches: permutations between Virtual Machines (VMs) present in the cluster (see Figure 3.3). Entropy monitors the cluster using external tools (such as Ganglia monitoring system[1]) and retrieves its current configuration. Then, it computes a new viable configuration and an optimized reconfiguration plan. This plan describes the sequences of transitions to perform (i.e., migrating, stopping/resuming and running VMs) in order to pass from the current situation to the new optimized one. It is interesting to note that Entropy uses a constraint-oriented programming approach to find optimized configurations. It takes as input a set of VMs and their CPU and memory requirements (current configuration), and computes a placement of VMs on nodes in order to maximize the cluster resource utilization.

Many self-adaptive software systems and autonomic frameworks have been implemented based on control engineering techniques – e.g., [CSWW04, DHP+05, TDBH09]. Self-managing capabilities are achieved in a system by taking an appropriate action when detecting particular situations based on information sensed from the environment or from the system's internal state. The functioning of any autonomic capability relies on a control loop that collects information about a system and its execution context and acts accordingly to achieve a desirable property or result. Tharindu and colleagues has published a systematic survey of such works [PCHW12].

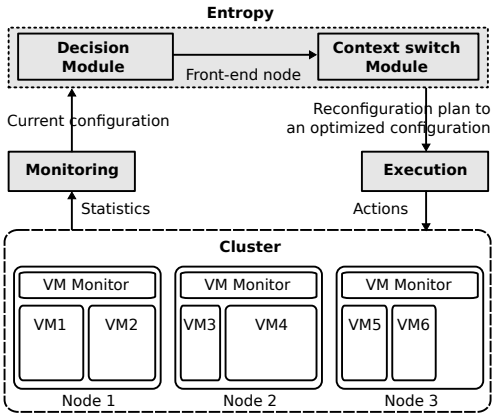


Figure 3.3: Architecture of Entropy

In addition, new component models such as Fractal [BCS09] or iPOJO [EHL07, EH07] provide this

[1] http://ganglia.sourceforge.net

notion of controllability at the low level of software components. Systems built using these frameworks can be observed and acted upon and hence they support the addition of control loops (embedded or external). More precisely, a Fractal component is made of two parts: a container (also called membrane), which embodies the control-related behaviour, and a content, which represents the functional code. An iPOJO component is also formed of two parts: a POJO object that implements the functional code, and a set of handlers that control different aspects of the component. In both component models, the controller membrane is connected to the target component using reflection. In iPOJO for instance, a particular handler controls the component's service references and changes the component state accordingly. Namely, if one of the component's required services becomes unavailable the handler sets the component's state to invalid; the component's provided services become unavailable as a consequence. When the handler detects the return of the required service, it connects the component to it and changes the component's state to valid; the component's provided services also become available. More advanced researches have been undertaken in this context to provide higher-level controllable systems based on the dynamic controllability features of such component-based model frameworks – e.g., [DBE08] and [PCT03].

Finally, an active research community from the control theory domain has joined the autonomic computing community. Therefore, since 2011, the International Workshop on Feedback control Implementation and Design in Computing Systems and Networks (FeBID) is being associated with the International Conference on Autonomic Computing (ICAC). Two years earlier, in his keynote at ICAC 2009, Joe Hellerstein [Hel09] outlined the different applications of feedback control to Autonomic Computing systems, such as memory allocation and throttling background utilities in commercial database management systems and thread pool administration.

3.1.3 Summary

Karl Astrom, a well-known contributor to control theory, stated that the "*magic of feedback*" is that it can create a system that performs well from components that perform poorly [Ast07]. This is achieved by adding the controller element that dynamically adjusts the behaviour of systems based on its measured outputs. Indeed, control theory is well suited to dynamic systems for ensuring some *performance* attributes such as response time or throughput. Foremost, it produces good results when the behaviour of the target system is well-modelled. However, it suffers from some disadvantages. In particular, it can create dynamic instabilities in a system.

Applying feedback control to individual autonomic elements is an interesting approach that has brought good results. However, there is still a need to better understand and control the behaviour of multiple interacting feedback loops. This is especially the case when the feedback loops to be integrated cannot be completely known in advance. In particular, when the managed system runs in an open environment with uncontrollable external services, feedback control loops are difficult to construct.

3.2 Agent-based systems

There is no universally accepted definition of the term *agent*. However, a software agent is usually defined as "*a computer system situated in some environment that is capable of flexible autonomous action in order to meet its design objectives*" [Woo97]. This means that an agent decides for itself what it needs to do in order to meet its specified objectives. It responds in a timely fashion to changes that occur in its environment. It can also exhibit a proactive behaviour by taking the initiative before problems occurred. Finally, most agents can interact with other agents and with humans in order to complete their problem-solving task. In such cases, agents are said to be *social*.

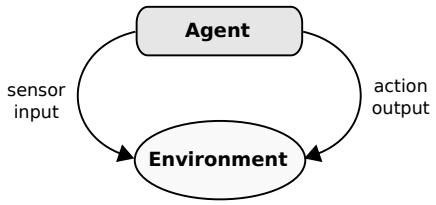


Figure 3.4: Computer System Agent in its environment

Figure 3.4 depicts a basic representation of an agent interacting with its environment [Wei99]. It takes sensory information from the environment and produces actions that affect the environment. The environment provides the conditions under which an agent can exist [OVDPFB03]. In other words, it defines the properties of the world in which an agent can and does function.

Beyond the capabilities of any single agent, *multi-agent systems* (MAS) employ multiple agents together to solve complex (distributed) problems. Jennings et al. [JSW98] summarize the characteristics of MAS as follows:

- Each agent is individually motivated and attempts to maximize its own utility;
- Each agent has incomplete information, or capabilities for solving the problem, thus each agent has a limited viewpoint;
- There is no global system control;
- Data is decentralized; and
- Computation is asynchronous.

3.2.1 Structural organisation of multi-agent systems

IBM's reference architecture for autonomic computing is very similar to the one depicted in Figure 3.4. However, in the research field of multi-agent systems, an agent interacts with its environment, which can include other agents. Hence, one of the objectives of multi-agent research is to find the best *organization* among software agents that can exhibit desirable performance characteristics [JSW98]. A number of organizational strategies have emerged from this line of research [HL04], like *hierarchical* organization (Figure 3.5-a), *team-based* organization (Figure 3.5-b), or *federation* (Figure 3.5-c). No one organization is suitable for all cases, but trade-offs can be found between the different organizations for specific situations. Most often, designers study the characteristics of each type of organization to determine under which situations it can be beneficial [CZ04, CZ06].

In the hierarchical organization, the agents are conceptually organized in a tree-like structure (Figure 3.5-a). Each parent agent can control its immediate descendant agents; it also detains a more global view than its descendants. To maintain such different degrees of visibility, low-level agents sense the environment and produce local data, which is forwarded upwards to higher-level agents and then aggregated with other sources to produce more global information. On the contrary, higher-level agents send control flows downwards when they have sufficient information to make a decision. This organization defines hierarchical authority levels between the different agents, where each agent is assigned a specific *role*. It follows the *divide-and-conquer* approach of decomposition that allows using a larger number of agents efficiently [YKO03]. This type of organization is strict and only allows agents to communicate with parent and direct descendant agents.

Abstracting the low level data can introduce uncertainty since important details may actually be lost.

Moreover, using a hierarchy can also lead to an overly fragile organization, prone to *single-point of failures* with potentially global consequences. However, a hierarchical organization reduces dramatically the computational load placed on any one agent, because the area that each agent is responsible for can be relatively small. Moreover, control actions become more tractable and increased parallelism can be exploited.

In the team-based organization, a number of cooperative agents agree to work together towards a common goal. Hence, each team tries to maximize its global utility (goal), rather than the goal of the individual members. The agents within a team interact in an arbitrary fashion, but which is consistent with the team's goal (Figure 3.5-b). Each agent can be assigned particular roles for addressing subtasks required to fulfil the team's goal. One of the benefits of such organization is that the group of agents can address larger problems than any individual is capable of. However, there is an increase communication among the team members to remain in coordination.

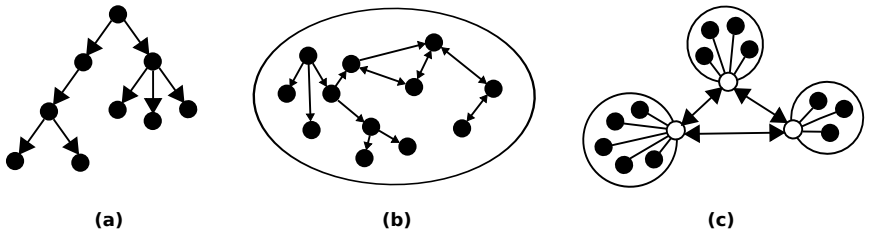


Figure 3.5: Example of agent organisations: a) hierarchical; b) team-based; c) federation.

In the federation organization, groups of agents, with a single delegated leader, have some autonomy to take decisions. Agents do not communicate directly with each other. Instead, they only communicate with the group leader, which is also called *broker* or *facilitator* [GF92, GK94]. The leader's role in this case is derived from and generalizes the concept of *mediator* [Wie92]). Group leaders cooperate with each other for taking high-level decisions (Figure 3.5-c). In fact, the agents from one federation give-up their autonomy to their group leader and the leader takes the responsibility for fulfilling their needs. This type of organization is comparable to a governmental system bearing the same name.

We note here that designers can mix different types of organizations for a specific problem. In [YKO03], Yadgar and colleagues provided such a mixed organization for distributed sensor environments. Groups of geographically related sensors are first formed into federations with a single leader acting as the intermediary. This federal organisation is then considered as the lowest level of an agent hierarchy. The leader of each federation collects raw data from the members of its group, and sends it to its parent agent in the hierarchy; this agent is known as a zone leader. It interprets the sensor data to the best of its ability and then forwards the interpreted data to its parent agent, which can in turn form a more abstract view.

3.2.2 Communication principles

Besides the organizational structures, several principles are also required in multi-agent systems to facilitate communication between the agents [OVDPFB03]:

Communication Language

To understand each other and cooperate, agents need to communicate using a common language. In particular, such language should define the type of messages that can be employed (e.g., assertions, queries,

replies, requests and denials) and the corresponding ontology. Examples of such languages include: FIPA ACL (Agent Communication Language Specifications) [FIP02], KIF (Knowledge Interchange Format), and KQML (Knowledge Query and Manipulation Language) [FFMM94]. Figure 3.6 shows a basic example of KQML messages in which an agent A sends a simple query to an agent B and receives a response via a tell.

(a)
Agent A sends the following performative to agent B

```
(evaluate
  :language KIF
  :ontology motors
  :reply-with q1
  :content (val (torque motor1) (sim-time 5))
)
```

(b)
Agent B replies

```
(reply
  :language KIF
  :ontology motors
  :in-reply-to q1
  :content (scalar 12 kgf)
)
```

Figure 3.6: Example of KQML message exchange

Interaction Protocols

A communication pattern between the agents may also be defined. This represents the sequence of messages between the agents and the constraints on the message content. Examples of such interaction protocols include contract net protocol [Smi80], Dutch auction protocol (also known as public bid auction) [FIP01] and Publish/Subscribe protocol.

Coordination Strategies

To achieve their goals, agents need to coordinate [Dur99], cooperate [Jen95], compete and/or negotiate [JFL$^+$01] in a shared environment, within the group in which they participate (Figure 3.7 provides a taxonomy of some of the different ways in which agents can coordinate their behaviours and activities).

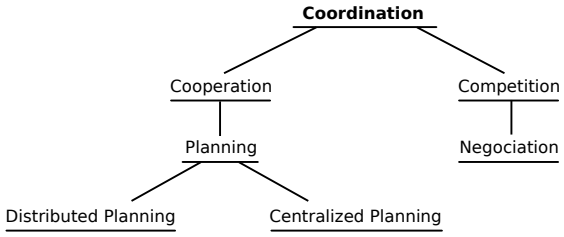


Figure 3.7: A taxonomy of coordination strategies [Wei99]

3.2.3 Adaptive multi-agent systems

Once a multi-agent system is initially deployed, there is typically a need for changes and adaptations to the system during its execution. This is also the case for most other types of software systems. In the context of our thesis, the main interest is in (1) the engineering of software agents so that they exhibit an adaptive behaviour and (2) the structural adaptation of the multi-agent organisation.

Agent-Oriented Software Engineering (AOSE) focuses on the definition of methodologies to guide the process of engineering complex software systems based on the multi-agent systems paradigm. Jennings [JSW98, JW00] promotes an agent-based approach to software engineering based on decomposing problems in terms of autonomous and decentralized agents that can operate with flexible and high-level interactions. Like in classical software engineering methodologies, very few of AOSE methodologies address the maintenance and runtime adaptability phase of software systems (or what is usually called *design for change*). One of the few AOSE methodologies that deal with this problem, in the context of multi-agent systems, is the Gaia methodology [ZJW03]. This methodology is focused on the use of organizational abstractions to guide the analysis and design of MASs. In [CZ06], the authors showed how Gaia could be used to model and tackle changes that can appear at runtime.

Runtime adaptation is a particularly demanded feature in pervasive computing systems. Such systems are characterized by heterogeneous hardware platforms, as well as by highly-dynamic execution environments, where devices can join or leave at any moment.. Hence, researchers trying to use agent-based systems to build pervasive systems were faced with the requirement of having adaptive agents in order to cope with recurrent changes. In [GZKL08], authors proposed an approach called VERSAG (*VERsatile Self-Adaptive Agent*s) in which agents are modelled as software components that have the ability to exchange their capabilities with peers, support multiple forms of adaptation (especially runtime compositional adaptation), and enable reuse as they are based on a component-based infrastructure. This architecture is used to handle pervasive computing in [GKLZ09]. The authors demonstrate the utility of adaptable mobile agents in such environments. Using a component-based infrastructure is also demonstrated to be viable for mobile adaptive agents in [AH05].

Several other research works investigated how the self-organization techniques can be applied to multi-agent systems to exhibit structural and compositional runtime adaptation [CZ04, SP08, PWB09]. Indeed, the authors of [CZ04] analysed the problem of modelling and developing multi-agent systems from the organizational theory point of view. In particular, they focused on the critical issue of adapting MASs organizations whenever changes in the overall system structure are required. They highlighted the role of principles like design-for-change, increments, modularity, and separation of concerns in facilitating the building of adaptable and adaptive MAS. In [SP08], the authors proposed an adaptive agent model based on self-organization mechanisms, and targeting small agent societies. Similarly, but for large-scale hierarchical topologies, the authors of [PWB09] proposed an adaptive agent-based self-organization approach for the robustness of self-adapting hierarchical topologies. In fact, as we have mentioned before, in hierarchical topologies, any local failure has a global impact on the global organization. To minimize the effect of such failures, Pournaras and colleagues proposed an approach called AETOS (Adaptive Epidemic Tree Overlay Service), which continuously tries to adapt the hierarchical topology to changes in the underlying environment in a *pro-active* and *reactive* manner.

3.2.4 Research work related to Autonomic Computing

A first workshop dedicated to agents for Autonomic Computing was held in 2008 in conjunction with the 5th International Conference on Autonomic Computing (ICAC'08). The aim of this workshop was to explore the potential of the agent paradigm and related architectures, models and technologies to contribute to the autonomic computing field. During the earlier years of the autonomic computing initiative researchers have also investigated the feasibility of a joint research between the two domains. Jeffrey O. Kephart highlighted this aspect in his Vision of Autonomic Computing article [KC03]: "*autonomic elements will have complex life cycles, continually carrying on multiple threads of activity, and continually sensing and responding to the environment in which they are situated. Autonomy, proactivity, and goal-directed interactivity with their environment are distinguishing characteristics of software agents. Viewing autonomic*

elements as agents and autonomic systems as multi-agent systems make it clear that agent-oriented architectural concepts will be critically important". Besides this conviction, IBM has developed a toolkit for building Multi-Agent Autonomic Computing Systems, which was called ABLE [BSP+02] (*Agent Building and Learning Environment*). ABLE toolkit is implemented in Java, and provides a JavaBeans library; a set of development and testing tools; and an agent platform. Via this toolkit, autonomic management is provided in the form of a multi-agent architecture. The different autonomic tasks are encapsulated and implemented via one or several interacting agents.

In [TCW+04], Gerald Tesauro and colleagues from IBM proposed another multi-agent system approach to autonomic computing. The developed tool, called *Unity*, is based on multiple interacting agents that ensure desired autonomic system behaviours, like self-assembly, self-healing, and self-optimization. Each agent manages one application in a shared cluster. It calculates the resource-usage utility for its application in a given environment. A high-level agent, called Resource Arbiter, computes a globally optimal allocation of cluster servers across all applications (See Figure 2.7).

In [CI09], the authors showed how to build a self-adaptive system based on a massive multi-agent platform for complex systems. Agents are generated automatically for managing dynamically the behaviour of the elements in the controlled environment. The proposed architecture is composed of two distinct layers: operational and morphological layers. At the operational layer, a set of Aspectual agents collects information from the controlled environment and forwards it, after interpretation, to the morphological layer. At this level (also called control layer), a set of Morphology agents create a view of the activity of the aspectual agents. Another set of Analysis agents interpret this morphologic space and try to exhibit specific characteristics by building plans using optimization functions.

Brazier and colleagues have established a research agenda for the possible integration of multi-agent systems, service-oriented computing and Autonomic Computing [BKPH09]. The shared objective of these three research domains is to cope with the complexity of today's computing systems. In particular, the authors outline the possible benefits of agent technology and service-oriented computing for autonomic computing: "*System-level autonomic behaviour arises from interactions among the autonomic elements, just as MAS behaviour arises from interactions among individual software agents. These interactions are dynamic and flexible in pattern (hierarchical, peer-to-peer, and so on); relationships among agents are established via negotiation and maintained via agreements created during the negotiation process. Agreements between agents and service providers are, in fact, SLAs*[2]*. Autonomic elements such as registries and sentinels play a role analogous to that of service registries and middle agents: to negotiate service provisioning as specified in the SLA*".

Furthermore, some recent works like in [Tal12] studied the potential synergies between multi-agent systems and Cloud-based systems. They investigated the applicability of agents for obtaining dynamic, flexible and autonomous behaviour in cloud computing systems, where agents can search, filter, query and update the massive volumes of data stored.

3.2.5 Summary

We pointed out that several agent-programming environments and toolkits have been developed to support specific agent architectures. Several interaction protocols were proposed, like 3APL [HDBVDHCM99], Cougaar [HW05], CORMAS [BBPLP98], JACK [HRHL01], JADE [BPR01], AgentSpeak [BPL04], Jason [BHW07], Madkit [GF00], and ZEUS [CNNL98]. Moreover, software engineering methodologies for analyzing and designing agent-based systems were developed, like Gaia [WJK00], Tropos [BPG+04], and AUML [OVDPB01]. Important work has been done to standardize some features of agent systems,

[2]Service Level Agreement

such as the one done with FIPA and KQML for inter-agent communication. While these environments, toolkits and methodologies were massively studied and applied to different traditional computing systems, a lot of work remains for adapting them to other emerging complex systems like those found in Grid, Cloud or Pervasive systems. The potential benefits of using an agent-based approach to self-manage these complex software systems are appealing; in particular, this approach exhibits a high-performance working load and high scalability as thousands of software agents can be employed to control large-scale software systems.

3.3 Nature-inspired approaches

Living natural systems constitute a particularly interesting source of inspiration for computer science researchers [HS07]. Indeed, biological systems exhibit complex structures and behaviours and often create them in a robust and efficient manner. In contrast, computer systems are becoming more and more complex and their development and management are becoming increasingly difficult and costly. Hence, researchers have started to look for analogies between computing and living systems and identify biology-inspired metaphors, models and mechanisms that can be applied to better understand, control and develop a new generation of computer systems [Mar00, FBGA02, Lod04, Nag04].

Similarly, natural systems were the initial inspiration for the field of Autonomic Computing. Indeed, in his manifesto, Paul Horn pointed out that autonomic computing should be similar to the human autonomic nervous system, which allows our brain to concentrate on higher-level conscious tasks by dealing with low-level internal functions autonomously [Hor01]. The main objective is hence to let the autonomic system anticipate and address administrative needs thus freeing human operators from managing complex systems by hand. This would allow human operators to concentrate on the business-level objectives that they want to accomplish instead [Mai02].

The initial biological models that inspired the autonomic computing metaphor feature interesting self-managing characteristics and properties. Examples of such properties include: *Scalability*, *Robustness*, *Self-Stabilization*, *Liveness* and *Safety*; in addition to *Regulation* of a targeted parameter or function of course.

In the following sections, we detail some of the techniques and primitives inspired from biology that are under active investigation and application in autonomic computing domain. The purpose of such efforts is to achieve the aforementioned qualities and properties along with the self-* capabilities in autonomic computing systems. Next, we introduce the self-organization and emergence phenomena and principles, which are also being studied actively to help increase the robustness and scalability of computer systems. The idea here is to distribute and decentralize the system's control across multiple fragments, hence eliminating centralised top-down control and its associated limitations (e.g. scalability issues, single-point-of failure and difficulty in developing all possible adaptations in advance). Subsequently, we will present some techniques for allowing the regulation of self-organizing behaviour. This leads to more deterministic behaviour or at least to behaviour that is able to obtain what is needed by system administrators. We conclude by outlining significant research works related to autonomic computing systems.

3.3.1 Self-organisation and emergence

Whereas the goal oriented inspiration to shift the control from users to the software itself influenced the autonomic computing initiative. Earlier works on nature-inspired metaphors focused on studying the collective behaviour emerging from the interaction of multiple individual elements. Emergence is a phe-

nomenon that describes the complex behaviour that arises spontaneously in many natural systems that comprise large numbers of interacting entities. Each entity exhibits rather simple behaviour and has no global system knowledge. The interaction of entities in a particular context results in global properties or behaviours that were not predictable from the individual behaviours (i.e. the whole is different from the sum of its parts). Self-organisation is a closely-related phenomenon where decentralised entities seem to organise themselves, without external control or direction. Here, organisation is considered in terms of increased structure or ordered behaviour. From a rigorous perspective, emergence and self-organisation focus on different aspects of the micro-macro interaction phenomena in decentralised autonomous systems [WH05]. Nonetheless, since these phenomena are often interrelated and appear together in complex adaptive systems, we use them interchangeably for the scope of this thesis.

Examples of self-organising and emerging behaviours include bird flocking and ant colonies. Here, the micro-level interactions of members create organisations or structures that are functional, stable and adaptable at the macro-level. The flock for instance is able to change direction and stay together when passing around obstacles without internal collision [SKPF03, OS06]. Similarly, in ant colonies, ants self-organize in order to gather food for the colony by passing simple local messages between them, either directly or via the environment (i.e. using a chemical substance called *pheromone*). Several algorithms have been inspired from ant colonies most often for implementing optimization search techniques or for the self-organization of computer networks [Sun11].

One of the contributions of this domain to autonomic computing is the introduction of emergence in the design of software applications (or what is called *emergent engineering [UD11]*). Indeed, Anthony demonstrated how emergence could add robustness and scalability to distributed autonomic applications [Ant04]. He focused on the design of distributed algorithms based on the science of emergence, allowing several elements to compete effectively for the resources they need.

Management and control in self-organizing systems are completely distributed - i.e. each participating subsystem has its own control process, as shown in Figure 3.8.

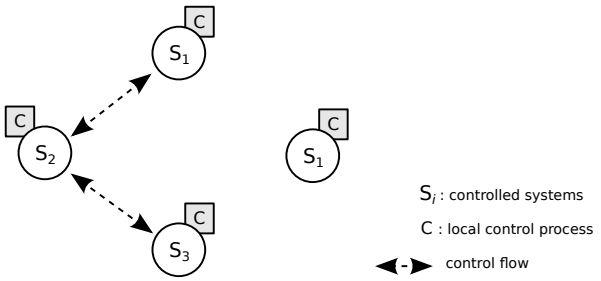


Figure 3.8: Decentralized control in self-organizing systems (adapted from [Dre07])

However, due to the complex, dynamic, and non-deterministic nature of autonomic self-organizing or emergent systems, a formal or analytic proof, or even an accurate prediction of the exact system behaviour is unfeasible. Research works like in [DWSHR05] provided an alternative approach based on advanced numerical methods, in particular the so-called “equation-free” macroscopic analysis, which allows analysing and assessing trends in system-wide behaviour. In the next section, we present some additional techniques allowing the regulation of the resulting structure and behaviour of self-organizing systems.

3.3.2 Functional structure and pattern formation

When Francis Crick and James Watson discovered the structure of DNA and its encoding schema in 1953 scientists could finally understand and explain a multitude of complex biological phenomena. The biologically encoded information (genetic instructions) in the DNA enables living organisms to develop into specific shapes providing particular functions. . This insight allowed scientists to develop several medical techniques and drugs for eliminating certain diseases and cure certain congenital defects in unborn children.

Certain analogies can be made between the DNA metaphor and computer programs, as both encode information that is used to control the behaviour and structure of a resulting executing object – i.e. molecules for living cells and biological organisms; and running programs for computers. Nonetheless, while the transformation of DNA (genotype) into organisms (phenotypes) is a complex, context-sensitive and decentralised process, the instantiation of traditional computing programs into executing applications is a rather straightforward process. This ensures a high-level of determinism for the resulting computing process but lacks some of the robustness, scalability and adaptability features of many biological processes. Hence, several research works set-out to push this metaphor farther and aimed to provide bio-inspired solutions for better dealing with the complexity of software system structure and dynamic adaptation requirements.

Amorphous Computing is one such related initiative. It aims to develop organizational principles and computer programming languages for building systems that feature coherent behaviours based on the cooperation of myriads of unreliable information-processing units [AAC+00]. The challenging problem that the Amorphous Computing community tries to solve comes from the fact that computers have always been constructed to behave as precise arrangements of reliable parts. Consequently, almost all the classic techniques for organizing computations depend upon this precision and reliability. However, new technologies in micro-fabricated particles or cells engineering (cellular computing) produce vast quantities of computing elements, which are relatively cheap but rather unreliable. Amorphous computing aims to capitalise on these properties. It strives to find a way to program such elements effectively in order to obtain a coherent behaviour despite the fact that they are unreliable and interconnected in unknown, irregular, and time-varying ways [AAC+00]. Indeed, in Amorphous Computing, the computing elements are placed irregularly and interact locally and asynchronously. They are possibly faulty, sensitive to their environment, and all programmed identically (since they are all fabricated by the same process). In general, they do not have any a priori knowledge of their positions or orientations.

Morphogenetic engineering is another interesting initiative that aims to explore the design and implementation of autonomous systems that are capable of acquiring complex structures and functions without central planning or external control [DSM11]. One notable work in this direction is [Dou11], in which René Doursat proposes a model that uses genetic-like information (coded as a set of context-sensitive rules) to regulate the differentiation and self-assembly of elements for generating patterns or more complex structures. Similarly, in [Bea11], Jacob Beal uses a functional blueprint to specify a system in terms of desired performance. The blueprint is then used to automatically grow composite systems with the desired properties, which are dynamically integrated and regulated.

3.3.3 Work related to Autonomic Computing

In [SH09], Roy Sterritt and Mike Hinchey applied biologically inspired concepts to provide an important requirement for autonomous and autonomic systems: *safety*. The authors drew inspiration from the *apoptosis* and *quiescence* metaphors observable in living systems to propose safety techniques in autonomic computing systems. The main idea is to introduce different types of signals between autonomic elements

in order to constantly verify their aliveness and well-being, and to trigger self-destruction mechanisms in expired or non-responsive components. A typical example of safety assurance in a biological context is that of a human hand touching a sharp object. This results immediately in a reflex reaction (hand being withdrawn) for getting the area in danger to a safe state (self-protection, self-configuration, and, if damage has occurred, self-healing). The self-destruction of cells that are either damaged or expired (apoptosis) is considered as a means of providing an intrinsic safety mechanism against non-desirable emergent behaviours (e.g. general infections or tumours). Similarly, autonomic computing systems should be equipped with appropriate self-destruction mechanisms to prevent them from developing unwanted behaviours or operating outside their designated timeframes. As we have indicated before, emergence can add robustness and scalability to autonomic computing systems [Ant04]. For instance, several autonomic elements can compete effectively for available resources that they require.

In an ongoing research work described in [Cap10], Caprarescu indicated how global management functions can be carried out via a self-organizing feedback-loop, inspired by the behaviour of ant colonies (see Figure 3.9). The system is composed of two layers: the first layer represents the components of the system and their interconnections; the second layer represents the self-organizing feedback–loop, which can self-repair the system's architectural failures in a decentralized manner. In the second layer, a set of agents travel and communicate with each other using digital pheromones; these are stored in the functional components of the system (in the first layer).

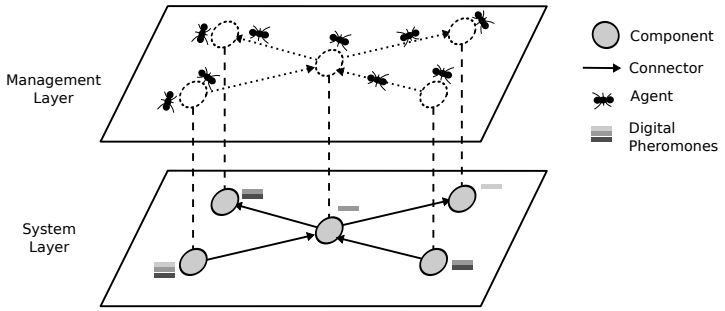


Figure 3.9: Autonomic architecture inspired by the behaviour of ant colonies

Finally, it is interesting to note that in 2006 a workshop dedicated to Engineering Emergence for Autonomic Systems was organized in conjunction with the International Conference of Autonomic Computing (ICAC'06) to study and discuss applications of emergence in autonomic computing, and the associated issues. The subjects of interest included interaction mechanisms; composition; contextual awareness; stability and behavioural scope in real-world contexts.

3.4 Architecture-based approaches

In the last decade, Software Architecture has developed into an important field of computing science and engineering, considered as an important tool for dealing with the increasing software system complexity [PW92, GS93, SG96, Gar00]. Hence, developing computer systems goes beyond writing algorithms and data structures, additionally encompassing high-level abstractions and organizational principles. Such architectural abstractions help the understanding and the design of the overall system, and serve as a communication conduit for different stakeholders involved in the development of such systems.

A considerable body of research work has been developed in this field with the aim of developing a technological and methodological base for treating software architecture as an engineering discipline

[Gar95, Gar00]. There has been a remarkable achievement in this field; companies now have software architects guiding the development of their software systems. A lot of software frameworks, architectural standards, methods, techniques, design patterns and production tools have been developed and used in real use cases. All these achievements have contributed to simplifying the understanding and development of complex systems by following recognized architectural styles like "*pipeline*", "*client-server*" or "*three layer*" architectures.

Besides, architectural models have been used as centrepieces for dynamic software adaptation. In order to effectively modify a system, an accurate model of its architecture must be available during runtime [Ore98]. A subset of the system's architecture is deployed as an integral part of the system. The deployed architectural model describes the interconnections between components and connectors, their mappings to implementation modules, as well as constraints on the different possible configurations. The mapping enables changes specified in terms of the architectural model to effect corresponding changes in the implementation. The runtime architecture infrastructure maintains the correspondence between the model and the implementation.

A number of frameworks and mechanisms have been proposed to enable self-adaptation. We highlight a few of them below. While they vary in details, the approaches share a number of common characteristics: they generally use a closed-loop control and an architectural model for reasoning about the target system; and they assume certain structures in the target system and adapt for a fixed set of quality attributes.

3.4.1 Software Architecture

Many definitions of software architecture have been provided in the literature. However, there is no single widely-accepted definition of the term. Many authors consider architecture as an abstraction of the system providing enough information to guide development, deployment and maintenance, but also as a basis for analysis, evaluation, and decision-making. Most of the commonly used definitions are consistent in their use of the terms 'component', 'connector' and 'runtime interactions'. In the following, we give the most relevant definitions.

According to Clements and Northrop [CN96], Software Architecture is:

> *"the structure of the components of a program/system, their interrelationships and principles and guidelines governing their design and evolution over time."*

That is, software architecture is a specification of how software components are structured to form the software system; this includes the type of components used as well as the constraint on how to connect them. A similar definition was also provided by Shaw and Garlan in [SG96]:

> *"Software architecture [is a level of design that] involves the description of elements from which systems are built, interactions among those elements, patterns that guide composition, and constraints on these patterns."*

This definition is strongly related to the definition given by the IEEE Standards Association [Com00], which define the word architecture as:

> *"The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution."*

Len Bass and colleagues adds more details to the previous definitions [BCK03]:

> *"The software architecture of a system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them."*

They consider the architecture to be all the different possible structures (or points of views) of the system. This implies that there can be different kinds of components (e.g. objects, modules, processes, libraries

or database) as well as interactions between the components (e.g. type of synchronization, data transfer or control). Each component is described through its "externally visible properties" which represent assumptions that other components can make about the component. This includes for instance a component's provided services, performance characteristics, fault handling, shared resource usage, and so on. Therefore, architecture typically focuses on views of a system's structure: how the system is split into elements, what are the externally visible properties of those elements, and what are the relationships among the elements.

3.4.2 Adaptation concerns

A software system can be modelled in several ways and from different views. Accordingly, there are many aspects of runtime adaptation and change; Oreizy and colleagues highlighted five important ones [OMT08]:

- Changes to the model behaviour;
- Changes to the model state;
- Changes to the execution context of the machine;
- Asynchrony of change;
- Implementation probes.

To achieve runtime adaptation of software systems, architecture-based approaches suppose that the managed systems support several adaptation operations [Ore98] like adding components, removing components, upgrading or replacing components, changing the architectural topology by adding or removing connections between components, altering the mapping of components to processing elements, querying for properties of architectural elements (e.g., to obtain versioning information), querying the current architectural topology and reconfiguring the different properties.

From an autonomic computing point of view, these adaptation operations can be qualified and used to support two main qualities: self-healing and self-configuration. Principal works using architectural models have investigated these two qualities. In [DvdHT02], Dashofy and colleagues investigated the use of architectural models to create self-healing systems. They provide an infrastructure that helps expressing and executing repair plans and reconfiguration actions when anomalies are discovered.

3.4.3 Enabling Techniques

Dynamic Architectures

In the early days of this field, software architectures were most of the time specified informally, often in terms of boxes and arrows, without precise associated semantics. This led to a number of incoherencies and ambiguities. Informal diagrams cannot be formally analysed to ensure consistency and verify the correctness and completeness of the proposed solutions. Architecture Description Languages (ADLs) were proposed with the purpose of providing an answer to this problem. An ADL allows the definition of a common architectural vocabulary between the different actors involved in the software development, from requirements specification to execution. Consequently, different tools can be developed to help parsing, displaying, compiling, analysing, or simulating architectural descriptions of software.

This constitutes one of the most active research areas concerning software architecture. Different ADLs have been proposed in the literature [MT00]. Each proposed ADL, in addition to representing architectural design, provides some specificity to handle the particular requirements of the domain where it is used. Interesting work was conducted by David Garlan and colleagues to develop a common framework

for the interchange and integration of different ADLs and their tools, called ACME [GMW97]. ACME was developed with the aim of providing support for the integration of different architectural tools, and hence including support for sharing architectural descriptions between them. Formally speaking, it is a common interchange format for architecture design tools.

Dynamic Architecture Descriptions provide languages to specify how architecture can change and evolve at runtime. Examples of such Dynamic ADLs include Darwin [MK96], Rapide [LKA+95], and Dynamic Wright [ADG98]. Darwin is an ADL that captures the dynamic structures of distributed software systems as the elaboration of components and their bindings in a configuration. Hence, Darwin permits the description of dynamic software architectures in which the organization of components and connectors may change during execution. This work is based on Pi-calculus to present the operational semantics. This is the same for other works such as in [Oqu04].

Architectural Styles

In order to provide common solutions for specific types of problems and to address various recurrent issues, several architectural styles have been identified and classified [TMO09]. An "architectural style" specifies how the architectural elements are used for creating one or more architectures in a consistent fashion. It can be thought of as a set of constraints on architecture – constraints on the component types and their interaction patterns. These constraints define families of architectures that satisfy them [BCK03]. In another words, a style represents a template for specifying the architecture of a specific system. Each architectural style focuses on certain quality attributes and provides its associated analysis methods and design tools. Hence, the selection of an architectural style for a particular system should be guided by the desired quality properties of that system.

Some self-adaptation propositions have considered the architectural style as an essential enabler of dynamic systems adaptation. In [CGS+02], Cheng, Garlan, and colleagues have used architectural styles to assist the dynamic repair of the running systems. The style for a given application should include along with the used architectural elements, the list of supported operators that can change the elements at runtime. Examples of operators for a web-based application running on a group of web servers include the adding of a server, moving servers between groups of servers, and/or removing a server. Secondly, the style also provides some information about the repair strategies written in terms of these operators associated with the style's rules. When detecting a constraint violation, the appropriate repair strategy will be applied to correct the error by using properties of the style.

There are several well-known architectural styles featuring specific design and runtime constraints. Examples include Pipe-and-filter, Data-flow, Client-Server and Publish-Subscribe. For each architectural style, some self-adaptation frameworks have been proposed in the literature. Weaves framework [GR91] is one of the earliest works that we know of. It supports dynamic rearrangement of components for applications that follow the data-flow architectural style. For the publish-subscribe architectural style Peyman Oreizy has investigated the evolution of software systems following this style via runtime changes to the architectural model, as modelled via the C2 framework [Ore00].

Model-based

In addition to architecture-based approaches dealing with the architectural view of a system other model-based approaches support and use design-time models to cover other aspects of the managed system. The use of design models at runtime offers new opportunities for autonomic actions without increasing development costs. This is accomplished by means of a planned reutilization of the efforts invested at design time [CGFP09].

Zhang and Cheng from Michigan University developed the "Rapid" tool that helps achieving self-adaptation by using formal models (i.e. Petri Nets) to model the behaviour of target software systems [ZC06]. Petri Nets are used at runtime to reason about the possible adaptation actions to take. In [MFB+08], authors investigated the use of aspect-oriented computing combined with model-driven engineering to support runtime adaptation.

Recent approaches propose to bring models into the runtime. These models are known as *Models@Runtime* [BBF09]. In this approach, runtime adaptation mechanisms that leverage software models extend the applicability of model-driven engineering techniques to the runtime environment.

Several research works have investigated this approach like in [Mao09, CGFP09, MBJ+09, GHT09, GIO09]. In [Mao09] for instance, Shahar Maoz proposed to use model-based traces as runtime models. From design-time sequence diagrams and state-charts, he extracted high-level events and used them at runtime to abstract low-level system execution of concrete programs. He also offered a partial family of metrics to analyse the generated models.

Variability

To exploit and reuse software artefacts (components, models, frameworks, documentation, and so on) across different versions of the same product or between different but similar products, researchers and practitioners have established variability models to ease and maximize reuse [CHW98]. Variability models help achieve this objective by designing the different variability points where developers can exchange artefacts without influencing the global architecture and objectives of the software system. In [CGFP09], researchers have investigated the use of variability models at runtime to have a rich semantic base for achieving autonomic behaviour during execution. In response to changes in its execution context, the autonomic system can use the variability model to exploit and choose an equivalent variant for replacing a broken or non-optimized component. It can hence plan and execute the necessary modification actions to the system architecture. That is, the variation model plays the role of a formal specification of management policies for guiding the autonomic system's adaptation at runtime. Figure 3.10 shows an example of the use of variability in smart-home applications. Variability is added to the feature model [KLD02], which specifies system functionality by means of incremental features. Features are hierarchically linked in a tree-like structure through variability relationships such as optional, mandatory, single-choice, and multiple-choice.

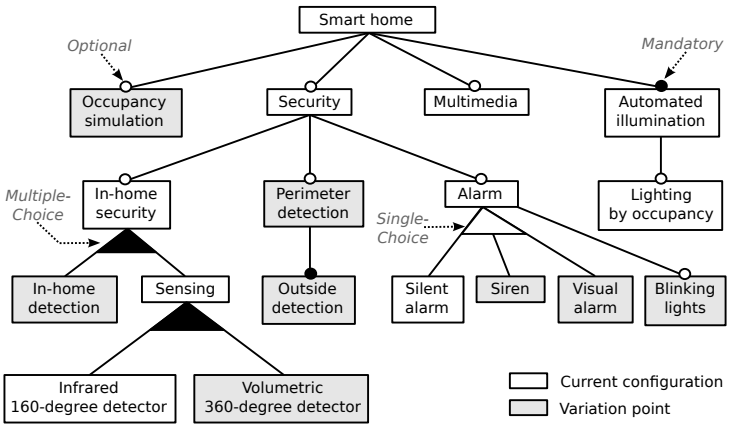


Figure 3.10: Smart-home feature model at runtime. Features are hierarchically linked in a tree-like structure through variability relationships.

In Figure 3.10, white boxes represent current features activated in the system, while gray boxes represent potential variants that may be activated in the future.

Variability management at runtime was also used in other domains like software product lines to allow dynamic reconfiguration at execution [HHPS08]. A Software Product Line (SPL) is defined as "*a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way*" [Ins13]. This well-defined role of each asset in an SPL leads to more flexibility when it comes to changing them systematically for building a new product (that has a similar architecture) or even for the same product at runtime. Typically, some variation points related to the environment's dynamic properties are bound at runtime. This architecture allows an easy reconfiguration since components can dynamically appear or disappear from configurations, and communication channels can be established dynamically between these components. In [BTRC05], authors have extended the SPL approach with extra-functional features and provided a constraint programming resolver to reason on it. This enables their approach to cope with the increasing complexities of variability.

Separation of concerns

Separation of preoccupations enables the separation between the application's business logic and the code that ensures non-functional properties, including the adaptation logic code. It has become an important principle in software engineering [CE00] in general, and especially in dynamic software adaptation. It captures the degree to which issues concerning a system's functional behaviour are distinguished from those regarding runtime change. The greater the separation, the easier it becomes to alter one without adversely affecting the other. The fact of using an external control-loop to self-adapt software systems is the best example of such principle.

Reflection

Reflection enables two main functions when used for dynamic adaptation of software systems [DSC+00]. First, it allows system analysis by queries on the system's structure and internal states, which then serves to create an adequate architectural model. Second, it allows system adaptation by injecting reconfiguration actions in the right places of the architectural model, which can then be reified into the actual system. These functions require a rigorous two-way connection between the system and its representation (also called causal relation). Tools and middleware dealing with reflection and self-adaptation should ensure rapid and correct synchronization between the architectural model and the actual software system it represents. This means that changes in the internal system structures should be reflected in the architectural model, and vice versa. Starting with the adoption of architectural reflection in programming languages, reflection has lately also been introduced into operating systems, virtual machines (VMs) like the JVM, and component-based frameworks like iPOJO or Fractal.

Reflection can be structural, as provided in iPOJO, which includes architectural models for the internal representation of a component or a composition. These are representations of the underlying system structure and facilitate adaptation by providing knowledge on the components and their interconnections. Reflection can also be behavioural, which enables by dynamic inspection of data-flow monitoring and manipulation, and access to resources and their management.

3.4.4 Summary of architecture-based approaches

Several dynamic architecture-based adaptation approaches have been developed to date, focusing on formal models and mechanisms of adaptation. However, only a few of them tackle explicitly the problem of control distribution and decentralization, and its consequent engineering challenges. Notably, engineering support should be available to help make a target system self-adaptive. Such reusable support can be provided in the form of integrated tools, which can help developers to more easily define adaptation policies that are more abstract and domain-specific in nature and that must take into consideration business goals and quality attributes.

The studied frameworks are strongly tied to the underlying architectural style of the target software systems. Architectural styles provide information about the possible adaptation operations on the system. Stateless data-flow systems are often explored as they exhibit natural and simple adaptation behaviours.

The main autonomic qualities investigated by researchers using architectural models at runtime are self-healing and self-configuration. The autonomic frameworks associated with such approaches generally monitor the target system in order to build an intermediate architectural model. An architectural model evaluator detects incoherencies and problems with respect to a reference model, and develops a repair plan. The plan is first executed on the architectural model and consequently applied to the target system.

Due to the growing size and complexity of software systems, scalability is an important issue. The management of reconfiguration in dynamic software architectures can be either centralized in a specialized component or distributed across decentralised components. In general, a decentralized or distributed approach is more likely to scale. Currently, the management used in most of the architecture-based approaches is centralized, not distributed. This is primarily because early types of dynamic architectural change such as ad-hoc and programmed often had centralized management. Newer definitions of change, as found in self-organizing architectures, use decentralised management in order to account for scalability in large-scale distributed systems [BCDW04].

3.5 Discussion and chapter summary

In this chapter, we reviewed some techniques and methods from existing approaches related to autonomic computing systems. Several techniques that we have incorporated in our proposition, as described in the next chapters, come directly from these domains. First, the control-based approach defines the principal idea of separation between the managed system and the control system. IBM's reference architecture can be seen as an advanced control loop. Advantages of such approach are multiple. Notably, it allows managing and controlling systems that do not already have this notion of self-managing properties embedded into their code. This includes legacy systems and modern systems. Also, it can facilitate the reuse of the autonomic control logic across several managed systems.

Agent-based approaches are actively investigated in several research communities. We are interested in how software agents can be organized to fulfil objectives. We have seen that each structural organization is suited to specific situations with particular requirements. Furthermore, we saw the communication principles between agents, including the communication language, interaction protocols, coordination strategies and social policies. Several research works investigated adaptive multi-agent systems.

Natural-inspired approaches were also presented in this chapter, and essential techniques and primitives were outlined. Most importantly, we showed the phenomena of emergence and self-organization, where individual elements featuring relatively simple behaviours and interacting with each other can exhibit interesting global behaviour. Another important technique presented is the decentralised formation of patterns and complex structures based on a DNA-related metaphor. This technique is based on the encoding of essential information that can then be used to reproduce, differentiate and construct complex adaptive systems.

Finally, we presented some advanced techniques coming from software engineering. We have focused on software architecture and model-based self-adaptation of software systems. We outlined the main techniques that help achieve system adaptation at runtime.

As a conclusion, we believe that these domains can be merged to build innovative autonomic systems with important quality attributes:

- Architecture can play an important role in "reuse", "analysis", "evolution", and "management".
- Multi-agents can play an important role in the structural organization of multiple autonomic managers (either cooperative or in conflict).
- Nature-inspired techniques can play an important role in policy specifications and self-organization of autonomic managers.
- Control-based approaches can play an important role in the application of self-management capabilities to legacy systems and in general to systems that do not exhibit self-management support. Control Theory expertise can also help address well-known feedback-loop problems including oscillations or divergence.

Chapter 4

Proposition

Contents

4.1 Problems addressed **55**
4.2 Contributions overview **57**
4.2.1 Positioning of the thesis within the team research 57
4.2.2 Thesis contribution 58
4.3 Objectives and Applicability **59**
4.3.1 Design decisions 59
4.3.2 Applicability assumptions and scope 60
4.4 Cube Framework overview **62**
4.4.1 Theoretical Support 62
4.4.2 Solution Overview 63
4.4.3 Coordination Strategy 64
4.4.4 Cube Autonomic Manager 68
4.4.5 System Management Life-Cycle 69
4.5 Summary **71**

In this second part of our manuscript, we present our approach for autonomic computing systems. We have developed a framework called CUBE that uses, in a synergistic manner, different techniques from the state-of-the-art (detailed in chapter 3) and provides novel architecture with important quality features that meet the requirements for administrating complex software systems. In this chapter, we detail the problems addressed and the objectives. We then present an overview of our approach that aims to overcome the shortcomings observed in the afore-studied autonomic computing frameworks (in chapter 2). The next chapter details the proposed architecture and provides more explanation about its functioning and internal modules (chapter 5). Later on, we detail how our approach is implemented and we provide more technical details about how to use and extend it for specific situations (chapters 6 and 7).

4.1 Problems addressed

Autonomic Computing aims to provide self-management capabilities to software systems, including self-configuring, self-healing, self-optimising and self-protecting functions. As we have seen in chapter 2,

IBM has proposed a logical, reference architecture inspired from control theory to develop autonomic computing systems. This reference architecture has fostered a number of valuable research works.

There are a substantial number of achievements in the autonomic computing domain in terms of self-managing functions implemented into computing systems. This is especially the case in enterprise and cloud computing but also in communications and pervasive computing. To achieve this, researchers have devised innovative autonomic solutions to address specific problems occurring in individual systems. Nonetheless, the larger and more difficult task of combining such focused solutions into complete autonomic systems must still be addressed [DSNH10].

Moreover, to capitalise on existing experience and results, the common principles and mechanisms implemented individually in specific autonomic solutions should be extracted, generalised and provided in the form of reusable frameworks for autonomic management systems. Such frameworks should provide a reusable base for addressing the aforementioned challenges of modern computing systems, including resource heterogeneity, dynamicity, increasing number of cooperating systems, and frequent changes in administrative requirements and execution conditions (see introduction chapter 1).

We have conducted a study of a representative list of the existing autonomic computing frameworks in subsection 2.5.2 of chapter 2. The resulting comparison, presented in subsection 2.5.3, showed that the studied autonomic frameworks lacked some important qualities for tackling the challenges of modern software systems. In the following list, we summarize the most important shortcomings:

Lack of scalability. Most of the studied autonomic frameworks use a centralized architecture to control software systems. The centralized (and often monolithic) control cannot be decoupled onto different execution nodes, even if the target system can be distributed on separate machines. That is, the control decisions are made in a single place. This has the advantage of having better decisional capabilities as all needed information is available in one place. It is also simpler to implement (than a distributed solution) since there is no need for synchronisation and coordination functions. However, this organization features serious scalability limitations and introduces a single point of failure. In addition, at present, it is still relatively rare for autonomic solutions and technologies to be deployed in production because of their hidden costs [Hel09]. For example, model-based approaches for self-configuration and self-optimization are powerful in laboratory demonstrations, but they typically incur high costs for model construction and maintenance. Autonomic computing frameworks that use architectural models maintain a runtime model instance of the entire managed system, including all the needed details; they reason on this model to take decisions. Hence, such frameworks use a translation infrastructure to maintain a coherent architectural model of the target system. When considering complex and large-scale software systems, building and maintaining such centralized architectural model is not feasible.

Lack of dynamic adaptation support. As the managed software systems are more and more dynamic and flexible, autonomic frameworks are also required to change with the managed systems and hence support dynamism and runtime adaptability. As it was defined in section 2.5.1, *dynamic adaptation* is the ability of software systems to add, remove or exchange components, modules, and/or services during execution. Moreover, when a system is considered open, the components or services introduced at runtime are unknown a-priori at system design time. Most of the studied frameworks support static or offline adaptability of some sort, but only a few of them support runtime adaptability.

Lack of support for heterogeneity and extensibility. Most of the studied autonomic frameworks are specific to particular domains. They are often delivered with specific sensors, actuators and control strategies. Using these frameworks for software systems in other domains, or based on new technologies is generally not supported. Only *Rainbow* and *Ceylon* are more generic and allow reasoning

on heterogeneous resources.

Lack of Software Engineering support. Very few methodologies have been proposed for engineering autonomic systems that systematically address requirements, design, implementation and assessment. To achieve most of the targeted quality attributes (subsection 2.5.1) developers of autonomic computing systems should, among others, adopt classic Software Engineering techniques, such as modularity, encapsulation and loose-coupling.

In addition to these important shortcomings we also include the following remarks:

Several autonomic computing frameworks use ECA[1] and Utility function-based techniques to specify and ensure user objectives. These techniques reach their limitations when used in large-scale environments. Managing hundreds or millions of heterogeneous events and take adequate actions is very complex task. This is the same also for Utility functions because the complexity of choosing best adequate solution after evaluating all possible adaptation strategies.

The majority of autonomic computing frameworks use an external control approach. This was also the initial choice of IBM for its reference architecture. Few research works have envisaged the use of mixed and container-based control [PLL+06]. In this case, the developer of the business code should also take into the account the non-functional and administration aspects. Such mixed and container-based approaches can have good results at runtime when used for specific technology; however it cannot be generalized to cover all sub-systems for two main raisons; firstly, developers cannot predict all the possible management actions of future components, and secondly, because the existence of legacy software systems that are an integrated and an important part of business software systems. External control is like to be more suitable for Autonomic Computing System.

Finally, in most autonomic systems, the monitored information is limited to runtime aspects such as CPU or memory utilization, and undertaken actions are often just parameter tunings. In modern software systems we should enlarge the spectrum of monitored and/or controlled elements to cover application components and their possible connections, as well as the execution environment.

4.2 Contributions overview

4.2.1 Positioning of the thesis within the team research

This thesis is part of a broader research project – called Cube [DL09, DL11, DDL12] – which proposes a generic approach to designing autonomic management systems that can address the aforementioned challenges – i.e. heterogeneity, dynamic adaptability, scalability and evolution of business objectives. The Cube approach promotes two core techniques:

- The use of an abstract architectural model to formally specify administrative goals; this helps control the autonomic management process and ensure the viability of resulting system configurations;
- The decentralisation of the autonomic management process in order to avoid a single point of control and the associated limitations to robustness and scalability; decentralised managers coordinate to obtain coherent systems that conform to the architectural model.

This approach promises to provide an autonomic solution that can address two key requirements that are apparently contradicting:

- Decentralisation and open adaptation to support scalable and robust system changes in response to unexpected execution contexts;

[1] Event-Condition-Action

- Control and determinism over resulting system properties and behaviours to ensure key business objectives.

The Cube approach offers a compromise between these antagonistic forces, where system designers may chose the most suitable trade-off between the two extremes case-by-case. Namely, more determinism can be achieved by limiting the variability allowed by the model specification (which becomes more concrete) and by implementing more deterministic coordination mechanisms among autonomic managers (which may be more time and resource consuming). Conversely, more runtime flexibility implies more abstract architectural models and less restrictive coordination mechanisms.

While this approach seems promising, it also raises difficult problems that must be addressed when implementing concrete autonomic systems. The most important questions include:

- How to design and formalise the architectural model? The model must be restrictive enough to ensure core properties of the managed system while also allowing sufficient variability for adaptation;
- How to split administrative tasks among decentralised autonomic managers? Dynamic changes in the managed system impose that task assignation to autonomic managers be adaptable during runtime;
- How to coordinate the activities of autonomic managers so as to ensure the coherence and conformance of the resulting system to the modelled objectives? Autonomic managers may have to coordinate both locally and remotely, to ensure local or more global system properties, respectively.
- How to ensure the extensibility of autonomic managers to support future business objectives and various types of managed resources?

4.2.2 Thesis contribution

This thesis contribution is positioned within the general Cube project. It consists of a concrete framework for facilitating the development of autonomic applications following the Cube approach. The challenge in providing this framework consists in providing viable solutions to the many difficult problems raised by the Cube approach (listed above).

In this thesis we have studied the aforementioned questions, identified viable solutions and selected a certain combination of design decisions for developing the proposed framework (chapter 5). The framework is particularly targeted for developing applications in the *data mediation* domain. The selected design decisions reflect this fact by relying on important assumptions specific to the data mediation domain.

For experimentation and validation purposes, we implemented a prototype of the proposed framework (chapter 6) for supporting a particular data mediation technology – Cilia Mediation [GMD+11] – and for addressing several administration scenarios in specific data mediation applications. The applications targeted involve remote medical surveillance and monitoring of home resource consumptions (chapter 7). The administrative scenarios supported include the initial deployment of the data mediation application and the self-adaptation and self-repair of the existing application instance at runtime. Experimental results indicate the viability of the autonomic system developed via the proposed framework, with respect to the general objectives defined by the Cube approach.

In summary, the positioning of the thesis with respect to existing research in the team can be indicated by splitting the Cube project into three main abstraction layers:

1. The Cube approach – this is a conceptual layer, defining the general idea for autonomic system design: controlling the self-organisation process of decentralised autonomic managers via abstract architectural models that formalise business objectives.

2. The Cube Framework for data mediation – this is a design layer, providing a reusable and extensible specification language, architecture and core functionalities for facilitating the development of autonomic systems that follow the Cube approach and target the data mediation domain.
3. The Cube Framework Prototype – this is an implementation layer, providing a prototype implementation of the Cube Framework, with specific extensions for goal specifications and management of concrete resources in targeted mediation applications; these were based on Cilia Mediation technology.

With respect to this conceptual layered organisation, the contribution of this thesis represents the second and third layers – a domain-specific Cube Framework and an application-specific prototype for Cilia mediation resources.

The remaining of this chapter introduces the thesis objectives and scope (section 4.3) and an overview of the proposed Cube Framework (section 4.4). For simplicity, the generic conceptual aspects and the framework-specific design decisions are presented simultaneously since they are both critical to the understanding of the proposed contribution. Also for simplicity, within the scope of this thesis we sometimes refer to the Cube Framework as simply Cube. Even if an initial Cube prototype has been developed previously, the only Cube Framework developed and maintained at present is the one we propose in this thesis.

4.3 Objectives and Applicability

4.3.1 Design decisions

The objective of our work is to develop a **novel autonomic computing framework**, based on solid software engineering techniques, in order to help developers and administrators to achieve self-management capabilities in simple and cost-effective ways. To remedy the shortcomings identified previously (section 4.1), we first adopt the following design decisions:

1. **Architecture-based**. We believe that software architecture provides an appropriate level of abstraction to describe and reason about software systems. Indeed, many software quality attributes can be achieved through architectural choices. In addition, architecture can play an important role when aiming to maintain critical system functions and properties while undergoing repeated adaptations and long-term evolution. Finally, architectural solutions can be applied to a wide range of domains hence increasing the reusability of our approach. When system architecture is represented at an abstract technology-independent level this approach also provides support for the heterogeneity and extensibility of managed elements.
2. **External autonomic manager**. Our approach relies on the reference architecture proposed by IBM where managed artefacts are decoupled from the elements that manage them (called autonomic managers). We believe that such organization provides better support for the evolution of managed and managing elements.
3. **Decentralized organization of autonomic managers**. We have seen in this document that centralized organizations, although simpler to design and implement, are limited in terms of scalability and robustness. For this reason, we believe that our framework should be composed of a set of decentralized and decoupled autonomous managers that collaborate to achieve global high-level goals. At the same time, to limit the delays and communication overheads specific to decentralised coordination solutions, we also introduce several semi-centralised mechanisms for handling the coordination among remote autonomic managers. We consider this mixed design provides a good compromise

within the targeted data mediation domain.

4. **Collaborative autonomic managers**. Collaboration should be used to achieve the specified objectives. Managers should cooperate rather than compete in order to achieve their goals, to the best extent possible. This approach is more likely to avoid undesirable emergent effects and unpredictable behaviours, as sometimes observed in self-organized autonomous agents [Ant04]. Certainly, this approach only applies when objectives are specified by a single, coherent authority (i.e. no conflicts of interest); otherwise, competition among decentralised managers is inherent in the business model they serve.
5. **Homogeneous knowledge**. In order to manage different sub-systems in a coherent manner, autonomic managers should rely on the same kind of knowledge, even if some managers may have more information than others.
6. **Limited size of knowledge**. For performance reasons at large system scales, we believe that (architectural) information regarding the system under execution should be carefully managed. For each autonomic manager, only the necessary information should be collected and maintained.
7. **Modularity and extensibility**. Our framework should provide support for developers to build specific modules and integrate them easily into the management system to handle specific administrative and technological aspects. This improves support for heterogeneity as technology-specific modules can be introduced as extensions to the autonomic manager. Modules should also be reusable for autonomic managers that administer similar aspects in different systems.
8. **Runtime Adaptability**. Given the ever-growing dynamicity of managed elements and execution conditions, autonomic managers should be able to dynamically use and instantiate the adequate management modules for a given situation and technology.

4.3.2 Applicability assumptions and scope

While addressing some of the shortcomings discussed before, these design decisions are quite structuring and, obviously, have a strong impact on the applicability of our approach. Although we are trying to cover a broad range of modern software systems, the scope of our work is limited to autonomic software systems that have the following characteristics:

Component-based software systems. Since we focus on architecture, our work more naturally applies to component-based systems. This is hardly a limitation since component-based approaches are widely used in modern computing. In the data mediation domain, the component-based architectures generally take the form of a directed graph, with vertices representing mediation components and edges the one-way communication links between them. Moreover, in the targeted experimental applications we only dealt with directed acyclic graphs (DAGs). The proposed framework should in principle also apply to cyclic graphs and bi-directional communication but such cases have not been studied or tested within the scope of this thesis.

Stateless components. When stopping, replacing or resuming the execution of a component, it is often necessary to ensure the transfer of state from the *old* component instance to the new one. This is a highly complicated issue, which is out of the scope of this thesis. In our validation, we assume that components are stateless or that a quiescence mechanism is available.

Well-connected, high-QoS communication network. The proposed framework was design based on the assumption that the managed data-mediation applications would be deployed and executed on distributed platforms connected via a communication network that is well-meshed and that features relatively high bandwidth and low latency characteristics (e.g. the Internet). This excludes for

example mobile ad-hoc networks where connectivity is sparse and expensive. This assumption was particularly important when designing the coordination mechanisms among decentralised autonomic managers.

Coherent authority. The presented approach only focuses on addressing autonomic management objectives specified by a single coherent authority. This implies that cooperating autonomic managers can reach a solution that is satisfactory with respect to the administrator's objective(s). Cases where autonomic managers belong to different administrative domains and therefore compete for attaining inherently-conflicting objectives are outside the scope of this proposal.

No need for global system optimisation. At the self-management level we have fixed some assumptions for our work. Most importantly, we assumed that for the targeted data-mediation applications survivability and adaptability over the long term are more important than global system optimisation. Hence, obtaining a global *optimum* solution is *not* among our main objectives. This means that we will not try to find all possible global solutions and then select the optimal one. Rather, we want to create an acceptable solution progressively, taking any possible optimization decisions at each step. Hence, the proposed cooperation process among autonomic managers guarantees neither a unique solution nor an optimal one. Instead it aims to find one solution that belongs to the viability space defined by the administrator's objectives and that is influenced by opportunities in the current execution context. The reasoning behind this choice is to favour scalability, quick adaptability and robustness, rather than global optimality.

With respect to the application domain and technology, we have evaluated our approach in the context of data-mediation systems implemented based on the *Cilia mediation framework* [GMD+11]. Cilia supports dynamically-adaptable components that can be added, removed, or changed in terms of connections at runtime without stopping the system. In Cilia, dependencies among components are data-based. This means that no interface verification is needed when assembling components; there is only a specification of the semantic and format of input data and output data. Also, Cilia implements a quiescence mechanism which ensures that the state of the system remains correct before, during and after a runtime modification.

The Cube approach raises many important and interconnected challenges pertaining to several research domains. Within the scope of this thesis we could only concentrate on addressing subset of aforementioned challenges. Yet, when taking design decision we tried to make sure that, at least in principle, these do not seriously hamper future framework extensions that may aim to address the remaining concerns. Some of the most significant aspects that remain outside the scope of this thesis are discussed below.

We have **limited the self-star properties** to be supported to three important ones: *self-instantiation*, *self-configuration* and *self-repair*. Within the general Cube approach, these properties are also referred to collectively as *self-growing*. Indeed, the same system self-growing mechanisms intervene to set the system in place initially, as well as to adapt and repair it during runtime. At this stage of our research, we have also partially applied the proposed solution for the self-optimisation of resource consumption of cluster machines, however we do not provide solutions for self-protection. It remains out of the scope of our thesis. For the time being, such additional properties can be indirectly achieved in our framework by limiting the system's viability space explicitly to configurations that are known to be optimal and/or secure.

4.4 Cube Framework overview

In this section, we provide an overview of our proposed framework, which implements the Cube approach. We start by outlining the theoretical support of our work and then we introduce the different elements of our proposal. In the next chapters, we describe Cube framework in more details, focusing on its implementation and its applications.

4.4.1 Theoretical Support

Our work draws inspiration from research results of domains related to autonomic computing, as detailed in the state of the art (chapter 3).

Regarding *control theory*, we studied how control techniques can be used to regulate dynamic systems that constantly interact with their environment. Controllers are most often developed to provide the following objectives: regulation, disturbance rejection and optimization. In our work, we focused on the first objective, since we aim to regulate the managed software application over time and against dynamic changes in its execution environment. The control objective (or reference input) in our case is specified via an abstract, reference system model; all managed applications must conform to this model. The main means of regulation is via the instantiation, configuration, interconnection or removal of software components. At the low level (as we will describe later), our framework consists of several control loops. Additional control loops can be included as needed to handle additional self-management concerns. The proposed framework structure is detailed in subsections 4.4.2 and 4.4.4.

Regarding *multi-agent systems*, we employed the agent paradigm to implement autonomic managers and achieve a decentralised control solution. We have adopted a federation-like organization, where autonomic managers are organized into groups with a dynamically-elected leader. Decentralised autonomic managers collaborate with their immediate 'neighbours' (defined later) to fulfil high-level objectives. Group leaders are in charge of ensuring more global coordination processes. Several other coordination strategies have been proposed in the multi-agent systems literature [HL04]. We believe that the particular form of federation-like organization adopted for our framework is the most adequate in our situation. It provides a good compromise between total decentralisation and more 'classic' hierarchies or centralised solutions. More precisely, autonomic managers act in a totally decentralised fashion for as long as their local interactions can lead to globally-acceptable solutions. In addition, when required system properties must hold over larger system areas, group-wide control is enforced via leaders. In cases where targeted areas or groups become really large, further decentralised solutions can be introduced to replace the group leader's functions. Yet, this case was not addressed in the presented work.

We were extensively influenced by *nature-inspired* techniques and concepts. Most significantly, developmental biology provided an essential paradigm based on the genotype – phenotype transformation. The genotype encodes key information that guides the organism's growing process and ensures that the most important objective is met: either the resulting organism features the core characteristics of its species, or the process terminates without producing any viable organism. At the same time, the genotype does allow for the growth process to be context-aware; and for the resulting organisms (phenotypes) to be adapted to their particular environment. Moreover, such adaptations can continue to occur all along the organism's life, in response to environmental changes; the same mechanisms intervene to provide self-repair functions (e.g. especially in the case of plant species). In our case, the abstract reference model specified to define business objectives (detailed in section 5.3 of chapter 5) plays the role of a genotype. It is replicated across all autonomic managers which contain identical core implementations. At runtime, each autonomic manager uses the reference model and contextual information in order to differentiate. This means that it will delimit a portion of the reference model which it will then 'express' and maintain,

resulting into a managed application part (phenotype part). Neighbouring autonomic managers are considered those that administer system parts which are adjacent in the reference model. They collaborate in order to join their application parts and form the global application (the entire phenotype). Here, global coordination mechanisms present in organic systems (e.g. area-wide inhibition based on diffusion and reaction processes) are replaced in our case by simpler, more efficient procedures based on group leaders.

Finally, to realize our approach, we have heavily relied on advanced techniques from modern *software engineering* disciplines like Software Architecture, Model-Driven Engineering and Service-Oriented Computing. First, we relied on the assumption that software architecture can be employed, to a certain extent, as a means of ensuring core system properties, including basic functionality and quality features. This consideration was used to define the abstract reference model, which specifies autonomic management goals. Namely, the reference goal specification takes the form of an abstract architectural model, defining the main architectural constraints (e.g. component types, cardinalities and interconnections) and quality-related attributes (e.g. minimum performance). Secondly, we adopted a Model-Driven Engineering approach and more recent *Models@Runtime* techniques to automatically manage software applications using the reference model, during runtime, in a context-aware manner. This is achieved via the collaborative work of autonomic managers. Last, but not least, we adopted the service-oriented paradigm as a means of enabling extensive runtime adaptability. Most importantly, service descriptions allow for the definition of abstract service types which can be matched with concrete service implementations during runtime. This matching process can be repeated at runtime, depending on the execution context (e.g. incoming client loads and available service providers). This aspect was adopted into the reference model. It allows administrators to define an abstract architecture (i.e. interconnected service/component types) that will be dynamically instantiated via concrete applications (i.e. interconnected services / component instances from various providers). In addition, service loose-coupling, late binding and hot-swapping properties (available in most service-oriented technologies) are key enablers of the proposed solution.

4.4.2 Solution Overview

Our proposal consists in an autonomic management framework for highly-dynamic, distributed and heterogeneous software systems. The framework is designed as a collaboration of *decentralized Autonomic Managers* (AM) (Figure 4.1), which are capable of self-organising in order to pursue a common goal. Autonomic managers are architecture-driven, in the sense that they pursue goals expressed as abstract architectural models; and maintain knowledge of the managed resources in the form of runtime architectural models.

More precisely, autonomic managers are driven by an architectural specification called *Archetype*. An Archetype represents the system's design objectives, which must be conceived so as to guarantee desirable system properties (e.g. functionality and quality attributes). An Archetype imposes a number of architectural constraints that must be met at any one time, while also leaving some room for architectural variation and hence runtime adaptation. It is an abstract specification where architectural constraints are expressed in terms of element types (rather than concrete implementations or instances) and relationships; and annotated with additional properties.

Using Archetypes to express management goals allows us to deal with two design objectives that are seemingly opposing. First, we'd like the autonomic system to be able to adapt to unpredictable situations, like dealing with erratic client loads, updates of the internal component implementations or various configurations of the underlying distributed platform. At the same time, we'd like to make sure that core system properties will always be achieved, despite the continuous changes and adaptations. Most notably, such properties include key system functionalities and minimum quality of service (QoS). The proposed

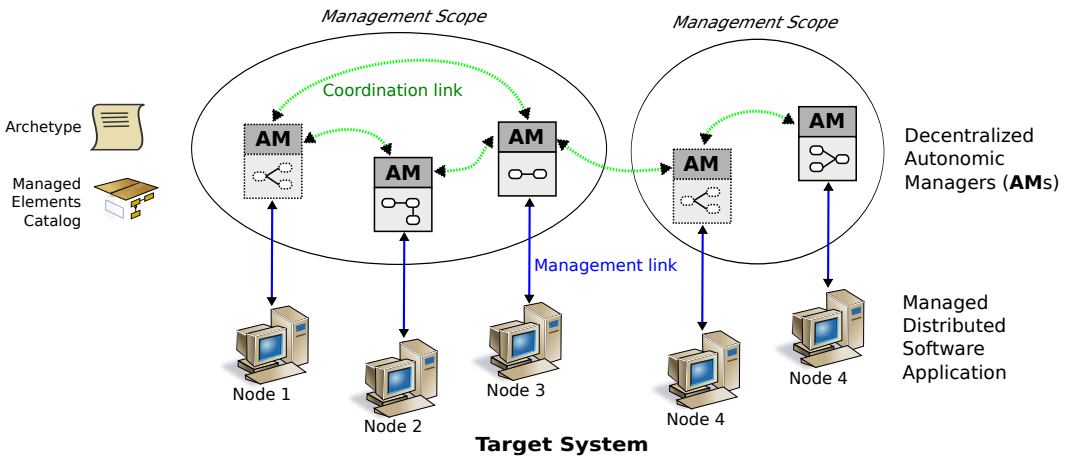


Figure 4.1: Cube solution overview

Archetype provides the right means of addressing both these concerns. On one hand, it specifies a rigid architectural template which must be implemented in all managed applications. System administrators must define this part so as to ensure the essential system characteristics. On the other hand, the Archetype also specifies a number of variability points which provide opportunity for system adaptation (bounded or boundless). System administrators must define this Archetype part so as to allow sufficient means of system adaptation to deal with encountered runtime situations.

In short, the Archetype defines the managed system's viability zone. All managed application instances must be within the viability zone. It is the role of the autonomic manager(s) to maintain the managed system within its viability boundaries.

When it comes to designing the autonomic management part for the framework we are faced with two additional design objectives, which are also seemingly opposing. On the one hand, scalability and robustness requirements demand a high degree of decentralisation in the autonomic management logic. On the other hand, ensuring a globally coherent behaviour and being able to reach system goals reliably seem to favour more strict control and a higher degree of centralisation. We address this challenge by introducing decentralised autonomic managers for dealing with the first aspect (scalability and robustness); and, special-purpose self-organisation and coordination mechanisms for ensuring the second one (coherence and global properties).

4.4.3 Coordination Strategy

Let us now briefly introduce the main self-organisation and coordination mechanisms. They rely on three main techniques:

- Having all autonomic managers pursue the same global goal, defined in the shared Archetype. Managers can split the goal into parts and manage one part each.
- Having autonomic managers coordinate their actions locally with their neighbours. Neighbours of a manager are those managers that administer goal parts that are adjacent to the manager's own part.
- Introducing semi-centralised coordination for autonomic managers that execute in well-defined areas, called Scopes, to ensure properties that must hold within those areas.

We shortly detail these three techniques next. More details are provided in the next chapter 5 "Cube Framework". The inner workings of each autonomic manager are briefly presented in the next subsection (4.4.4).

Shared Archetype

All autonomic managers share the same Archetype; an Archetype replica is distributed to each manager. This means that all autonomic managers in a Cube system collaborate to attain a shared goal. We defined a special-purpose Archetype definition language for formally specifying each Archetype (subsection 5.3 in chapter 5). As previously indicated, we do not address here cases where autonomic managers serve different authorities and must hence compete for achieving inherently conflicting goals. Treating such cases is complementary to our proposal.

To achieve the common goal, autonomic managers must split the Archetype into complementary parts where each manager will then administer one such part. Several alternatives can be conceived for the Archetype splitting process. For instance, one autonomic manager can be assigned to each component type in the Archetype. In this case, an autonomic manager administers all instances of the assigned component type, deciding when to instance them, how to configure them and so on. Here, the manager's neighbours are the managers administering component types that are connected to the manager's own component type, in the Archetype. Other possibilities include assigning an autonomic manager to each component instance [DL11] or to predefined Archetype parts. Finally, to deal with runtime unpredictability, we propose in this thesis that the Archetype splitting process be carried-out dynamically [DDL12]. It will hence depend on the distributed platform available and on the context-sensitive requirements of the managed application. As runtime changes occur, the Archetype can be re-split to a certain extent to accommodate the new situations.

In the selected design, each autonomic manager is assigned to one *Node* of the distributed platform, where a Node can represent either a physical or a virtual machine. Nodes are generally typed, depending on their physical resources, runtime load, security functions, and so on.

All autonomic managers have an identical implementation. This makes them easily replaceable and replicable, which is an important advantage for system robustness and adaptability. In the current framework implementation, autonomic managers are pre-instantiated on available system Nodes. At the same time, future extensions can easily be conceived to automatically discover new Nodes and integrate them into the managed platform by replicating and deploying a new autonomic manager on top of them.

When instantiated and started, an autonomic manager first evaluates its context, then reads the Archetype to determine the constraints that apply to this context. If no constraints apply it waits for a change. Otherwise, it starts administering the application running on the local Node so as to make it fulfil the constraints. For instance, a manager executing on a Node of type 'Mediation server' identifies an Archetype statement that imposes the presence of at least one 'messaging server' component on a node of this type. The manager becomes responsible for pursuing this constraint, which is consequently included in its goal part. If an instance of a message server cannot be detected locally, then the autonomic manager will instantiate one; this partial goal is hence met. In turn, this changes the manager's local context and triggers another evaluation of Archetype constraints that apply in the new situation. For instance, message servers must be connected to at least one component of type 'message destination. The manager must thus acquire a component instance of that type and connect it to its message server. If the component instance is created locally then the manager also adds the associated constraint to its goal part. When the autonomic manager can no longer solve applicable constraints locally, its goal part it fully defined. At that point, the autonomic manager would have differentiated its function so as to pursue that goal part. Also, the manager must now collaborate with neighbouring managers in order to continue the process and

resolve the remaining Archetype.

Local Collaboration

Autonomic managers must collaborate with their neighbouring peers, in order to join their partial management solutions into a global application that fulfils the overall goal (or Archetype). This functionality is mainly used for the self-instantiation of the distributed application. Namely, each autonomic manager creates and interconnects locally as many component instances as it can, considering the Archetype constraints and its own execution context. To extend the local application instantiation farther and to connect local instances to components executing on remote Nodes it must collaborate with neighbouring autonomic managers.

Neighbouring peers are determined dynamically, depending on the current context and on the applicable Archetype constraints. In the previous example, let's consider that the manager in charge of the messaging server cannot instantiate the 'message destination' component locally. This can be because another constraint imposes that this type of component can only run on a Node of type 'Web server'. Hence the manager executing on the 'Mediation server' Node will first look for a manager executing on a 'Web server' Node. It then asks this manager to take over the Archetype constraint which it could not resolve locally. If the 'Web server' manager accepts then it becomes the neighbour of the 'Mediation server' manager. The two managers are now collaborating to achieve complementary parts of the overall Archetype.

This process is repeated recursively, extending incrementally across autonomic managers until the entire Archetype is split and the overall application managed. When the current application meets all applicable Archetype constraints the management process pauses. If a change occurs later on the autonomic manager(s) detecting it are reactivated. For instance, an increase in the incoming client load can determine the autonomic manager on the 'Mediation server' to reconfigure the messaging server in order to comply with a performance-related constraint. As a more complicated example, consider that a component of type 'message source' is manually instantiated on a Node of type 'Gateway'. The autonomic manager executing on the 'Gateway' Node detects this change. It consequently determines that an Archetype statement imposes that all 'message source' instances must be connected to a 'message server' instance. Since message servers can only be found on Nodes of type 'Mediation server', the manager finds the autonomic manager that executes on such node and collaborates with it. The source message instance is consequently connected to the message server. To further complicate the example, we can now add a quality constraint that imposes a maximum number of incoming connexions into the message server. When messaging server reaches the maximum number of message source connections the autonomic manager administrating it will refuse any further connections. Hence, managers of newly created message sources will need to find other managers of message servers to collaborate with. The new message sources will consequently be configured to use another messaging server, installed on a different 'Mediation server' Node.

When two autonomic managers have been cooperating to validate an architectural constraint (e.g., connecting two remote components), they maintain a logical connection between them. At any time, if one of the distributed managed elements in either execution platform changes or disappears, its autonomic manager notifies the other autonomic manager about the change. The notified autonomic manager can then re-evaluate the architectural constraints involving the managed element that changed, and if they are no longer valid it tries to find a new solution.

Another interesting case involves the failure of a Node that is included in the management process. In this case, the autonomic managers executing on neighbouring Nodes detect the problem, look for an autonomic manager executing on an equivalent Node and try to collaborate with it. If successful, they would have managed to redistribute the Archetype and self-repair the managed system.

Scope-level Collaboration

As exemplified above, local collaboration allows decentralised managers to split and resolve various types of Archetype constraints. Nonetheless, other constraint types may require more global collaborations that stretch beyond direct manager neighbourhoods. A frequent case occurs when an Archetype clause sets a bounded cardinality for a component type. Let us change the previous example slightly and consider that one and only one instance of the 'message server' component can be available in each network domain, on a Node of type 'Mediation server', at any one time. Yet, several Nodes of type 'Mediation server' exist in each domain, for reliability reasons. In this case, the autonomic managers running on 'Mediation server' Nodes must collaborate in order to decide which one of them should instantiate the message server; and include the associated constraint in its goal part. However, these managers do not a-priori know each other; they are not neighbours since they do not collaborate for resolving complementary Archetype parts.

Various decentralised solutions could be introduced to address this coordination issue, based for example on peer-to-peer gossiping, message-diffusion and inhibition, and so on [BJM05a]. Nonetheless, in this thesis we decided to adopt a centralised solution for these precise coordination points. This is because we assumed that the number of managers involved in ensuring each such property will be much smaller than the total number of managers; and that the communication and processing load on the centralised coordinator will be quite reduced. This is particularly the case in the application domain we focused on during this research – data-mediation applications running on distributed platforms connected over the Internet. Here, communication is rather efficient and processing resources relatively abundant. Specific self-repair solutions must be added for ensuring the reliability of centralised coordinators (outside the thesis scope). On the contrary, a decentralised approach would have been more appropriate in application domains where communication and processing resources are more erratic and scarce, such as in ad-hoc (mobile) sensor networks.

Let us see how the proposed coordination approach functions to ensure properties over specified system areas.

First, to delimit targeted areas clearly we introduce a new concept called *Scope*. A Scope represents a well-defined section of the managed system. It can be defined as a physical area via geographical coordinates; as a network domain via an IP suffix; as a group of Nodes of predefined types; as a group of Nodes detaining certain resources; and so on, depending on the application domain.

In the Archetype, most constraints are defined with implicit Scopes that can be determined dynamically and via local coordination. For instance, the previous example stated that any message server must be connected to at least one message destination, which must run on a 'Web server' Node. Here, any 'Web server' Node can meet the constraint and the actual Node selected can be discovered at runtime.

For other constraints, the Scope over which they apply must be stated explicitly. Another example we showed above stated that one and only one 'message server' component can be available in each network domain, on a Node of type 'Mediation server'. Here, the constraint is defined explicitly over a Scope type that represents network domains; a Scope instance is one network domain.

Second, once Scopes are formally defined in the Archetype, they can be dynamically managed and employed for larger-scale coordination. At runtime, autonomic managers are automatically organised into Scopes, depending on the characteristics of the Nodes on which they execute. For instance, all autonomic managers that execute on 'Mediator server' Nodes will belong to a Scope type that has been defined to regroup all Nodes of type 'Mediator server'. Similarly, all autonomic managers that execute on Nodes belonging to a particular network domain will be regrouped into an instance of the Scope type 'network domain'. We can see here that several Scope instances can exist for the same Scope type. Also, in principle, an autonomic manager can belong to several Scopes at once.

Each Scope instance will have a dynamically elected *leader*. A scope leader has different roles. First, it maintains the list of all the Scope members, or autonomic managers. On the arrival of a new member, the scope leader saves its URI. Second, and most importantly, the scope leader helps manage scope-wide constraints. It represents a central coordination point, or arbiter, for decentralised managers that must reach a scope-wide property while acting in parallel and with no prior knowledge of each other. Let us see how a scope leader can help resolve the component cardinality constraint exemplified above, where one and only one component instance was allowed within any network domain. Here, all autonomic managers in a network domain are regrouped into one Scope instance, with one elected leader. When any of these autonomic managers tries to resolve the constraint, it first asks the Scope leader for permission. If it is the first manager to ask, the permission is granted and the manager is allowed to instantiate the component. The constraint is now resolved. If other managers, unaware of this fact, subsequently try to resolve the same constraint within this Scope, their request to the Scope leader will be denied; no extra component instance will thus be created.

4.4.4 Cube Autonomic Manager

Once the boundaries of their respective Archetype parts are well-defined, autonomic managers follow a rather 'classic' model-oriented approach to achieve them; that is, to set-in place and to maintain application parts that conform to their Archetype parts. This implies actions such as component deployment, instantiation, configuration, interconnection, and so on. Each autonomic manager builds a *Runtime Model* of the managed elements they administer. This runtime model represents one part of the entire runtime model of the managed application. Namely, a *local runtime model* reflects the state of the application part which is administered by the local autonomic manager. To reach its goals (Archetype part), the autonomic manager compares its local runtime model part to its Archetype part. In case of a deviation the autonomic manager sets-out to find a new application configuration that meets the constraints defined in the archetype part. It then triggers the necessary adaptations into the managed application part. This process may involve new collaborations, both local and scope-wide.

The internal architecture of a Cube autonomic manager follows the reference MAPE-K design. The internal constituents are organised into a control loop structure, which uses the runtime model and the Archetype as knowledge. The most important constituents are as follows:

A *Knowledge* module, which includes the entire Archetype and a *Runtime Model* (RM) part built and maintained during execution. The local RM only contains information necessary for the local autonomic manager to determine whether or not the local application part meets the high-level objectives (as specified by the administrator in the Archetype). This is an essential aspect for ensuring the scalability and performance of the autonomic management system. The local RM also includes modelled connections to the adjacent RMs; that is, to the local RMs of neighbouring autonomic managers. This enables special-purpose autonomic functions to monitor the well-being of neighbouring managed elements; or to be notified of any changes in their state.

A *Constraint Resolver*, which resolves (or finds a solution to) the manager's Archetype part. Namely, the resolver makes sure that the local RM part satisfies the constraints specified in the manager's Archetype part. Hence, the resolver module provides the control decision logic that ensures the correct functioning of the managed application part. To allow for the Archetype definition language to be extended, the constraint resolver is designed as a generic core extensible via constraint-specific plug-ins. For each new constraint type defined in the Archetype, a goal-specific resolver must be implemented and plugged-into the core resolver.

A number of *Technology-Specific Extensions* (TSE) including monitors and executors for specific technologies. Technology-Specific Extensions (TSEs) implement the causal relation [SHC+11] between the managed application part and the local RM. This means that runtime modifications in the managed elements are reflected into the RM (via sensors and monitors) and modifications in the RM are reflected into managed system (via actuators and effectors). Special-purpose monitors are also provided for sensing the environmental context; and the well-being of neighbouring managed elements via their representation in adjacent RMs. A set of TSEs must be implemented and plugged-into the framework for each of the implementation technologies of managed elements (e.g., TSE1 handle application components and TSE2 the messaging server). However, all TSEs update the same RM, which is an integrated abstraction of all managed elements. This allows the framework to be generic with respect to the managed application technologies and hence to be reusable across numerous platforms.

A *Communication* module to support the collaboration between autonomic managers (both local and scope-wide). Cube framework offers a generic interface to implement and use several communication technologies for its collaboration protocols like point-to-point communication via Sockets or publish-subscribe communication using a MOM (Message Oriented Middleware). In our developed prototype, we have implemented a point-to-point communication module (chapter 5, section 5.4.6).

In short, the control process proceeds as follows. Monitors collect data from technology-specific sensors and update the local RM accordingly. The Archetype Resolver is notified of any such RM updates. It checks the conformity of the new RM to the Archetype, takes the necessary decisions, and indicates any corrective changes by modifying the local RM. Actuators apply such changes into the managed elements concerned. More information about the internal architecture of Cube Autonomic Manager is given in section 5.4 of chapter 5 "Cube Framework".

4.4.5 System Management Life-Cycle

Figure 4.2 depicts the four main activities involved in our approach to system self-management. The first two activities are performed manually and off-line; they involve application-specific knowledge extraction and specification, as well as code writing and compilation. The remaining two activities are automated and performed during runtime; they rely on the control process introduced above and require little or no human intervention.

The actual steps are presented here after. The purpose of the **first step** is to model the system elements that have to be self-managed (e.g., software components, connectors, execution platforms and servers). These elements are modelled using an *architectural meta-modelling language*, which is provided with the Cube framework (section 5.2 of chapter 5). The core Cube's framework defines four domain-specific modelled elements: **Component**, **Node** and **Scope**. Components represent managed application modules providing various processing functions. Components are deployed on distributed platforms (or Nodes). Nodes are organized into various Scopes (or groups).

We notice here that administrators can model other specific system elements based on the Cube's Meta-model (section 5.2.2 of chapter 5) or by extending ones provided in the framework core. For instance, when targeting a specific application domain, the core Component concept may be insufficient for specifying all the important aspects to manage in an application component. In this case, the administrator can extend the core Component artefact and introduce a new one that provides the extra information.

Also during the first step, along with the modelled managed elements and their relationships, administrators should implement the necessary Technology-Specific Extensions (TSEs) (if not already done).

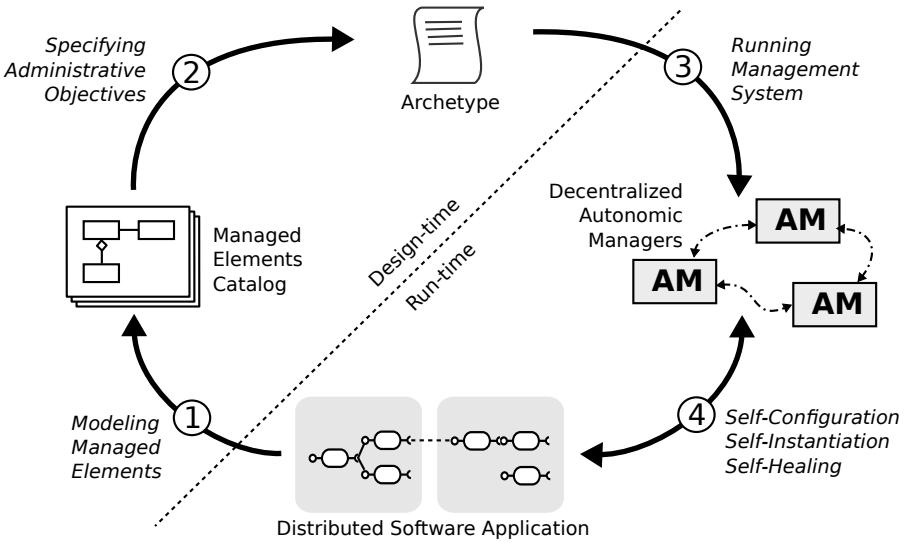


Figure 4.2: Life-cycle of Cube-based autonomic management process

These should correspond to the specific technologies implementing monitoring and execution functions specific to the elements of interest in the execution environment.

During the **second step**, administrators specify self-management objectives as a set of architectural constraints and quality attributes on the modelled managed elements. To do so, they rely on the models defined in the first step. Hence, management objectives are formally specified in the form of a declarative architectural specification (Archetype). Within the scope of this thesis we assume that the Archetype is sufficiently abstract to require no further update during the system's lifetime. Updating the initial Archetype remains a perspective of our work, which will be discussed in the last thesis chapter (8).

The **third step** consists in distributing the Archetype to all autonomic managers and bootstrapping the management layer. Hence, each autonomic manager receives a copy of the Archetype. Upon initialisation, autonomic managers cooperate to form the management Scopes (as defined in the Archetype).

Finally, during the **fourth step**, each autonomic manager monitors its managed application part, resolves local problems, and coordinates with other autonomic managers (either locally or at the Scope level) to resolve non-local administration problems. At this step, the Cube system operates autonomously and does its best for finding solutions to all administrative objectives. An autonomic manager can be both *reactive* – only executing in response to changes in the managed system; and *proactive* - imposing architectural configurations onto the managed application part, in order to fit the objectives defined in the Archetype.

4.5 Summary

In this chapter, we briefly presented our contribution to the self-management of highly-dynamic, heterogeneous and large-scale software systems, running in unpredictable environments. We outlined the main challenges faced when administrating such systems and we proposed a novel management solution for addressing these challenges. The most stringent difficulties associated with system management include system heterogeneity; high dynamicity in the system's internal make-up and execution environment; increasing numbers of integrated subsystems; and the ever-growing number of business requirements that evolve over time. We highlighted some of the main limitations of some of the most important autonomic management frameworks developed in the domain. The most important weak points in existing works include lack of scalability and the associated performance degradation; lack of support for adapting the autonomic management logic; lack of extensibility for supporting the heterogeneity of managed elements in various domains; and lack of software engineering techniques enabling modularity and reuse. We indicated how our proposal addresses the identified management challenges, while bringing key improvements and new features with respect to state-of-the-art management solutions.

Namely, using architectural models allows for the right level of abstraction in knowledge representation and reasoning. It provides common grounds for the integration of heterogeneous technologies and their associated evolutions and extensions. Moreover, this type of knowledge representation and processing mechanisms can be reused across various managed platforms. Separating the autonomic management logic from the underlying managed system also improves reusability and independent evolution of the two parts. Decentralising the autonomic management logic ensures better system scalability and robustness. The fact that each autonomic manager only maintains a partial architectural model during runtime also contributes to administration performance, especially in large-scale and highly-dynamic systems. The proposed combination of local and scope-level collaborations among autonomic managers ensures overall application coherence while minimising global communications which could severely impact performance. Finally, the modular and plug-in based design of autonomic managers and architectural model languages improve their reusability and extensibility for various application domains.

We presented an overview of our approach and introduced the internal architecture of our autonomic manager. Our solution is based mainly on the use of architectural models, interpreted during runtime by a set of decentralised autonomic managers. In Cube, architectures are used to provide abstractions of the managed elements of a system, their relationships and constraints. In particular, they are used to define goals via a formal specification called Archetype. Each Cube autonomic manager receives a full copy of the Archetype and proceeds to accomplish a well-defined part of it; Archetype parts are determined dynamically and may change over time. To achieve its local goals, each manager constructs a local Runtime Model of the local administered system and verifies it against the Archetype. The manager's Archetype Resolver performs such verifications, determining whether the monitored state of the local application satisfies all the Archetype goals. If this is not the case, adapted solutions are found and implemented, locally or remotely.

More information about the Cube approach; the autonomic manager's internal functioning; the global life-cycle; and Cube's application and validation in concrete use cases are detailed in the next chapters.

Chapter 5

Cube Framework

Contents

5.1 Goal and Knowledge Specifications 74
5.1.1 Overview 74
5.1.2 Key model-based management concepts 75
5.2 System Modelling Language – SML 76
5.2.1 ECORE 77
5.2.2 Cube Meta-Model 78
5.2.3 Domain-Specific Managed Elements 80
5.2.4 Cube Runtime Model 82
5.3 Goal Description Language - GDL 83
5.3.1 GDL Elements 85
5.3.2 GDL Properties 85
5.3.3 Archetype 87
5.4 Cube Autonomic Manager 89
5.4.1 Internal Architecture Overview 89
5.4.2 Runtime Model Controller 90
5.4.3 Archetype Resolver 92
5.4.4 Life Controller 96
5.4.5 Monitors/Executors (Technology-Specific Extensions) 98
5.4.6 Communicator 99
5.5 Summary 101

In the previous chapter, we presented an overview of our solution for the self-management of distributed and heterogeneous software systems. We introduced the global approach of the Cube project and the associated framework proposed in this thesis. In this chapter, we provide more details on the Cube framework and highlight its main contributions. In short, Cube framework provides reusable and extensible support for specifying goals and runtime knowledge via formal modelling languages; and a core architecture and implementation for the Autonomic Manager, which uses these models to adapt the managed system. The collaboration among decentralised Autonomic Managers is achieved via a special-purpose communication module included in the architecture.

First, we give an overview of the proposed Cube specifications for goal and knowledge representations (section 5.1). Then, we detail Cube's system modelling language - SML (section 5.2), and Cube's goal description language - GDL (section 5.3). Next, we present the internal architecture of a Cube Autonomic Manager (section 5.4) and provide details on each of its internal modules. We show how most of these modules can be reused and extended so as to facilitate the adoption and customisation of the Cube framework across various targeted applications.

The implemented prototype of the Cube framework is presented on next chapter 6, while the validation of our framework in national and regional research projects if detailed on chapter 7.

5.1 Goal and Knowledge Specifications

5.1.1 Overview

Cube requires modelling for two specific purposes. First, Cube formalises the goals of a managed system via an archetype, which takes the form of an abstract architectural model. Second, Cube represents the state of a managed system via a runtime model, which takes the form of a concrete architectural model. At runtime, the role of Cube's autonomic management functions is to ensure that the system's runtime model conforms to the archetype. In this section we focus on the modelling languages that Cube framework provides to enable this approach. For clarity here, we make abstraction of the Autonomic Managers' decentralisation and the fact that the system's runtime model is actually split among them. The presented languages and principles will apply unchanged to the decentralised case.

The two types of models – archetype and runtime model – require different definition languages. The Cube framework defines two such formal languages as follows:

- System modelling language (SML) - for modelling domain-specific elements;
- Goal description language (GDL) - for describing user objectives in the archetype.

In the following subsections we detail these two language types and explain the strong relation between the specifications they produce. In short, domain experts define specific managed elements based on the system modelling language (SML). The resulting element specifications are then used to represent the system's state during runtime, in Cube's runtime model. This model represents essential system knowledge that the Autonomic Manager uses for system administration. System administrators use the goal description language (GDL) to specify management objectives, in Cube's archetype. Objectives are specified as formal constraints over the system's architecture or quality properties. Notably, at this point, administrators can make use of the managed element specifications, defined previously via the SML, as parameters for the management constraints, defined via the GDL.

While designing Cube's specification languages for goal definition and system modelling, we have taken into account the following considerations:

- It should be possible to model any part of the managed system, as well as any relationships between these parts;
- Instances of modelled system elements are representations of the real managed system elements. To minimize the size of the model and simplify its interpretation, modelled element instances should only reflect properties that we want to manage, rather than all the details of the actual managed element.
- Modelled element instances should be coherent - they must all share the same meta-model, irrespectively of what they represent;

- Goals should be specified directly on the modelled system elements with simple coherent and interpreted language.
- At runtime, we should have the possibility to navigate between the modelled element instances, inspect them to learn about the system state and modify them to adapt the system. That is, the managed system should be reflective.

In the following, we first introduce some key model-related concepts, which we then use to detail the goal and system modelling specifications proposed in the Cube framework.

5.1.2 Key model-based management concepts

Simply put, modelling represents the activity of constructing models. The concept of model is not new, several definitions were proposed in the literature. Kleppe et al. defined it as "a description of (part of) a system written in a well-defined language" [KWB03]. By well-defined language the authors mean "a language with well-defined form (syntax), and meaning (semantic), which is suitable for automated interpretation by a computer" [KWB03]. From this definition, we understand that a model is a representation and an abstraction of a real system, and that a model should be written using a well-defined language which helps automating some processing tasks on the modelled elements. In addition, any operations on the model are considered to be effective on the system itself. Accordingly, Bezinvin and Gerbe refer to a model as "a simplification of a system built with an intended mind. The model should be able to answer questions in place of the actual system" [BG01].

In short, a model is a description, an abstraction, a representation or a simplification of a system written in a well-defined language. To facilitate their interpretation and manipulation by computers, models must be clearly and formally defined. The concept of meta-model is introduced to define the language used to express the model [OMG13]. Hence, the model must conform to its meta-model, which guarantees that the model is correct syntactically and suitable for automated interpretation and processing.

Several modelling languages and frameworks have been defined, including for instance UML[1], MOF[2], XMI[3], and ECORE[4]. Most of these are general purpose and some can be specialized for specific domains. In our thesis, we adopted one of these existing modelling languages - ECORE – as a standard base for our system modelling language. We then extended it with additional modelling layers, in order to provide more domain-specific modelling elements and hence facilitate the use of the Cube framework.

Loading and using models during a system's execution is part of a relatively recent research niche called Models @ Runtime [BBF09], where models are not only used at design time, but are also made available during runtime, to provide more knowledge about the system. Following this idea, several Autonomic Computing frameworks have proposed using models as a base for self-adaptation mechanisms [Mao09, CGFP09, MBJ+09, GHT09, GIO09]. Here, different modelling languages are best suited for various types of tasks [ZC06]. Software Architecture is seen as a special kind of model, one that is most suitable for representing software as abstract functional blocks and their interconnections (see section 3.4 from the state-of-the-art chapter 3). Cube adheres to this approach, placing architectural models at the core of its management solution.

More precisely, to manage component-based software systems, Cube relies on two types of architectural models. First, an abstract architectural model – the archetype – defines the management goals. Second, a concrete architectural model – the runtime model – describes the system state during execution.

[1] Unified Modelling Language

[2] Meta Object Facility

[3] XML Metadata Interchange (XMI) is an Object Management Group (OMG) standard for exchanging metadata information via XML

[4] ECORE is a meta-model for describing models with the Eclipse Modeling Framework (EMF)

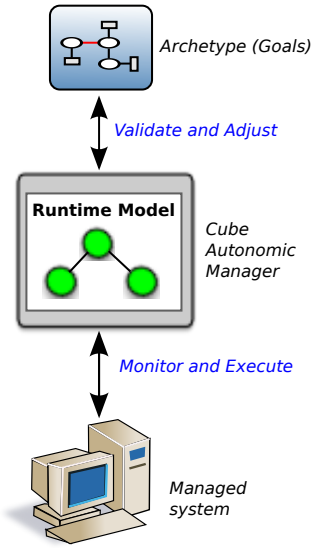


Figure 5.1: Overview of Cube's model-based management

Cube's Autonomic Manager constantly compares the runtime model with the archetype, verifies that the system's state conforms to the management objectives and takes action to adapt the system when this is not the case (figure 5.1). Hence, the archetype represents an Autonomic Manager's objectives. The runtime model represents its system knowledge, which is necessary for pursuing those objectives. The same applies in the decentralised case, where each Autonomic Manager performs the above procedure to a single part of the managed system and to the corresponding part of the archetype.

5.2 System Modelling Language – SML

To allow developers to make their systems self-managed using the Cube framework, the first thing to do is to model the system elements so that they can be handled by Cube Autonomic Managers (detailed in section 5.4). The Cube framework provides the *System Modelling Language* (SML) for this purpose. When designing the SML we wanted to achieve two things. First, we wanted to rely on an existing modelling language, such as UML or ECORE, in order to capitalise on the expertise inherent in its design. Second, we wanted to be able to model specific system elements, such as those required by the Cube approach and by various application domains. To achieve both of these goals we introduced a layered design for Cube's system modelling support (figure 5.2). Here, a model in one layer must conform to the modelling abstractions defined in the layer above. We can say that the model is defined based on the language provided in the layer above.

In brief, ECORE modelling language was adopted to represent the top-most generic layer (M3). This generic language can be used to model elements that are specific to the Cube approach, in the layer bellow (M2). Further bellow, the System Modelling Language (SML) extends the Cube-modelling layer with domain-specific elements (M1). The SML layer is situated above the runtime modelling layer (M0), meaning that SML is used to describe elements in the runtime model. The runtime model represents the bottom-most layer. In the rest of this subsection we provide more details on the constituents of the System Modelling Layers proposed in the Cube framework. We further detail each modelling layer and discuss the inter-layer relations in dedicated sub-subsections.

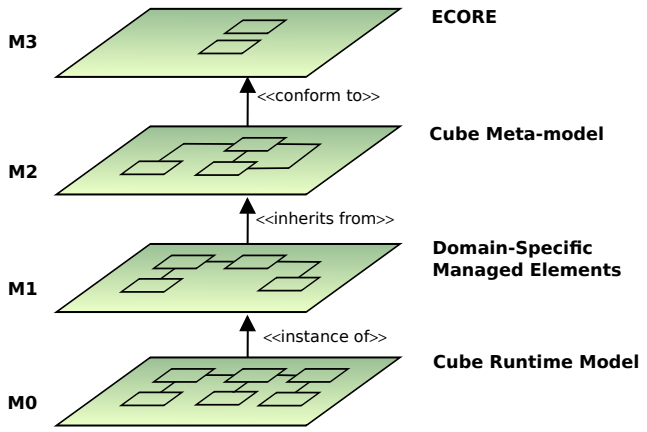


Figure 5.2: Cube framework's Modelling Layers and Languages

The Cube framework defines the concept of "Managed Element" to represent any managed resource in the targeted system. This generic concept can be extended to represent more specific system constituents that we want to manage, ranging from fine-grained software components to large-scale software systems like databases or application servers. The Managed Element concept is defined in the M2 layer - "Cube Meta-model". This layer is defined using the ECORE language (or meta-model). ECORE (M3 layer) is a part of the Eclipse Modelling Framework (EMF) [emf13]. Other meta-modelling frameworks could also be used to fit this requirement.

The Cube Meta-model layer (M2) is the most important layer in the modelling stack. It can then be extended in the M1 layer below with "Domain-Specific Managed Elements" so as to define more precise abstractions of the actual managed elements, such as Components, Nodes and so on. At runtime, Autonomic Managers create and maintain instances of such domain-specific model elements in the runtime model, in order to represent the real managed elements. This corresponds to the M0 layer - "Cube Runtime Model". This layer may include several modelled instances of domain-specific managed elements – e.g. several instances of the same modelled component, or several instances of different modelled components. Yet, all modelled instances have the same internal data structure that conforms to the M2 layer – Cube Meta-model – even where they are instances of model elements defined in M1 layer – Domain-Specific Managed Elements. Indeed, model elements in the M2 layer inherit from the Manage Element concept of the M2 layer (as detailed further in section 5.2.3).

All the modelling layers are handled by the Cube framework and are implemented in the prototype. The only layer extensible by users is M1 - "Domain-Specific Managed Elements" - for modelling specific elements in each managed system.

5.2.1 ECORE

To define the Cube Meta-model (M2), we need a meta-modelling language (M3) that can enable us to write the model specification, and that can also help us to generate code directly from the designed model. Despite the existence of several modelling frameworks like MOF [HIM+98], we have decided to use ECORE as a basis for defining Cube's Meta-model. This is motivated by the availability of associated tools that simplify the use and manipulation of meta-model concepts and provide a code-generation facility. ECORE is part of the Eclipse Modelling Framework (EMF) project [emf13]. EMF is a modelling framework and code-generation facility for building tools and applications based on a structured data model. The data

model is specified in XMI (a standard for exchanging metadata information via XML) and conforms to the ECORE meta-model. EMF provides tools to produce a set of Java classes for the model, a set of adapter classes that enable viewing, command-based editing of the model, and a basic editor. It also provides a run-time support for the ECORE models including change notification, persistence support with default XMI serialization, and a very efficient reflective API for manipulating EMF objects generically.

The ECORE meta-model includes the following concepts (Figure 5.3):

- **EClass**: represents a class, with zero or more attributes and zero or more references;
- **EAttribute**: represents an attribute, which has a name and a type;
- **EReference**: represents one end of an association between two classes. It has a flag to indicate if it represents containment and a reference class to which it points;
- **EDataType**: represents the type of an attribute - e.g. `int`, `float` or `java.util.Date`.

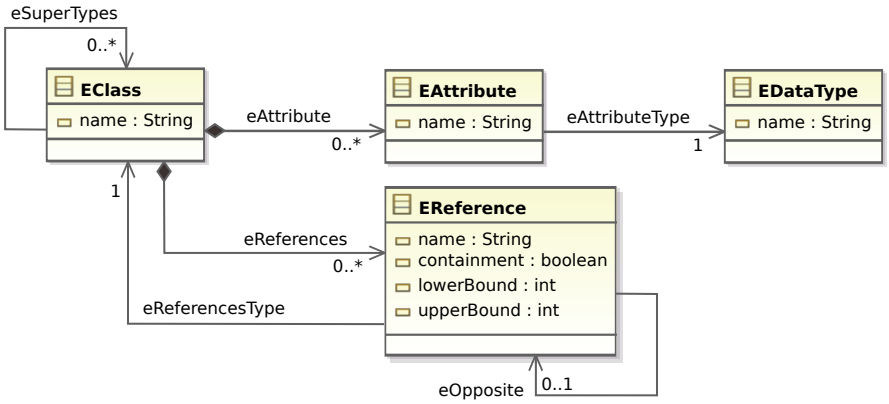


Figure 5.3: ECORE Meta-model

In the next sub-sub-section, we describe how the ECORE meta-model (M3) was used to define the Cube Meta-model (M2).

5.2.2 Cube Meta-Model

The Cube Meta-model layer (M2) uses ECORE to define modelled elements that are specific to the Cube approach. We propose the abstract model (or meta-model) depicted in figure 5.4 to represent any managed element handled by the Cube framework.Cube Meta-model defines the following concepts.

- **ManagedElement**: represents any managed element of the system. Its attributes are listed below. Some of these attributes – i.e. name and namespace – should be specified when customising the ManagedElement class (in M2) to represent a domain-specific managed element (in M1). Other attributes
  - **uuid**, am, and state - are initialized by the Cube framework at runtime.
  - **name**: the name of the managed element, like Component, Node, and so on;
  - **namespace**: represents a container for a set of managed elements;
  - **uuid**: the Universal Unique IDentifier of the managed element;
  - **am**: the URI of the Autonomic Manager where this managed element is hosted;

- **state**: represents the runtime state of this managed element's instance. Each Managed Element can have a set of Properties and a set of References to another Managed Elements.

- **Property**: represents any functional or non-functional property that we want to take into account as part of the self-management operations of the managed element. This may include for instance state values like cpu or ram consumption, or configuration values like the maximum inputs of a server. A property has a name and a value.
- **Reference**: represents any possible association between two managed elements. Any new domain-specific managed element that we model in the layer below (M1) should add all its relations with other managed elements as references. No direct Java references are allowed. Notice that a reference must have a name. Also, a reference can be unary – meaning that the current managed element can have at most one referenced element using this reference; or multiple - meaning that the managed element can have several references with the same name (e.g. the set of output components of one component).

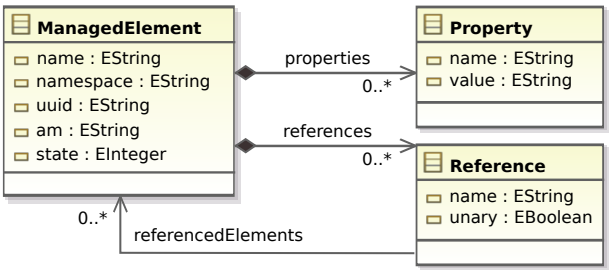


Figure 5.4: Cube Meta-model

As we have mentioned before, we used ECORE to specify our Cube Meta-model. The following figure 5.5 details an example of the relationship between a part of the Cube Meta-model and its corresponding ECORE model instance. Here, the ManagedElement class in Cube Meta-model corresponds to an object of the EClass class in the ECORE model instance, having its "name" attribute set to "ManagedElement". ManagedElement attributes correspond to objects of the EAttribute class of ECORE. In the example shown in the figure we show only two attributes – "name" and "namespace" – that have the type EString. The Property concept of the Cube Meta-model corresponds to another EClass object of the ECORE meta-model, with its "name" attribute set to "Property". The association "properties" between ManagedElement and Property concepts in M2 layer corresponds to an object of the EReference class of ECORE.

Using EMF allowed us to generate Java code directly from the Cube's meta-model, since this was specified based on the ECORE meta-model. The generated code has very interesting features, including a change notification mechanism and serialization. In our prototype implementation, we have slightly changed the generated code to fit our Framework needs. In particular, we have omitted the direct Java references between modelled element instances (e.g. references between modelled objects). Instead, we have used the Universal Unique Identifier (UUID) to represent the references between different modelled instances. This allows moving the modelled instances between different Java Virtual Machines without changing the entire model.

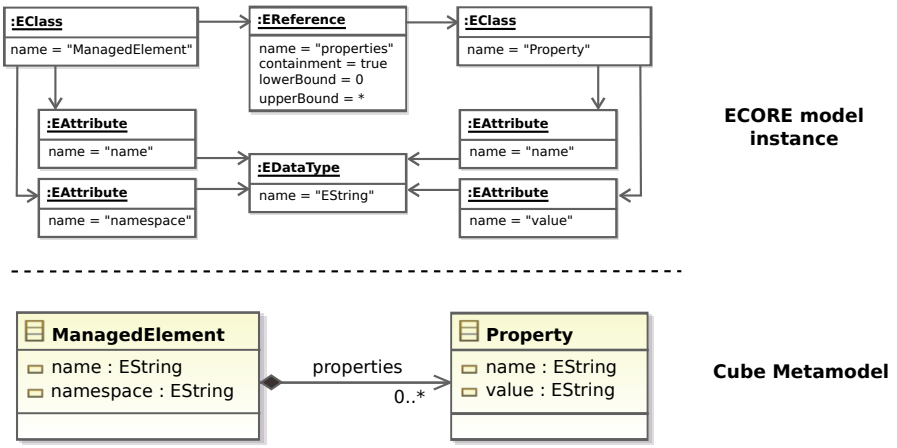


Figure 5.5: Cube Meta-model fragment and its ECORE model instance equivalent

5.2.3 Domain-Specific Managed Elements

The System Modelling Layer (M1) allows users to define their specific model elements for their particular managed system. Specific managed elements should be specified based on the Cube Meta-model, using its three main concepts: ManagedElement, Property and Reference.

In the developed prototype, we focus on distributed data-mediation systems (as described in the validation Chapter 7). In this specific domain, applications – also called mediation chains – consist of a set of components that are distributed across different execution platforms, or nodes. Such mediation components are interconnected so that data can be transported from its sources – at the bottom of the mediation chain – all the way to its destinations – e.g. servers, at the top of the chain. In-between data sources and destinations, each mediation component – also called Mediator – is responsible for a specific data-processing task. Hence, the general role of the mediation chain is to process and transport the data between its sources and destinations.

Self-management tasks in this domain can include the self-instantiation of mediation components and their placement on the available execution nodes, as well as more advanced functions such as balancing the load between distributed nodes.

Since Cube framework targets the mediation domain, we proposed a "core" domain-specific model to abstract and represent mediation systems. The following description and the remainder of this thesis will use this core model to describe and explain the different framework parts.

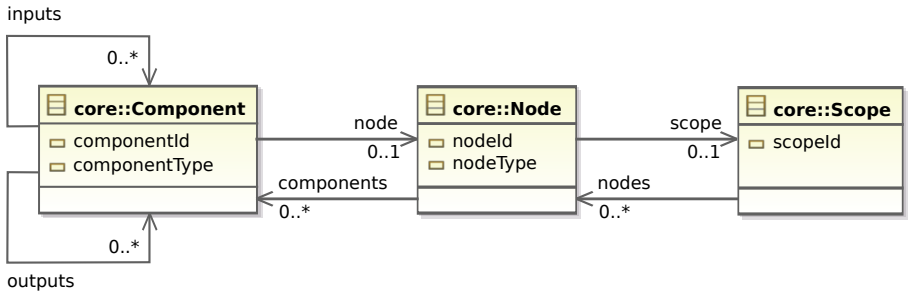


Figure 5.6: Cube framework's Core Domain-Specific Managed Elements

Figure 5.6 shows the core domain-specific managed elements we propose. We detail each element in the following list:

- **Component**: represents an abstraction of a software component. It has two properties: `componentId` – the identifier of the component instance; and `componentType` – the component type. These two properties are set at runtime when instantiating the Component model element. The Component has three references: inputs and outputs, which represent input and output references to other components; and node reference, which holds the UUID of the node where the component is hosted.
- **Node**: represents a node – a deployment platform or physical device – where component instances can execute. A Node has two properties: `nodeId`, which holds the identifier of the Node instance; and nodeType, which holds the Node type, like "PC", "Smartphone" and so on. The Node element has two references: components, which holds the list of references to its hosted components; and scope, which holds a reference to the Scope of which the Node is a member.
- **Scope**: represents a virtual grouping of Nodes (section 4.4.3 of the previous chapter 4). For instance, it can represent an administrative network domain, a geographical location, or a set of platforms sharing certain characteristics. The Scope element has one property: `scopeId`, which holds the scope identifier. It also has two references: nodes, that holds the list of the scope members; and master (not showed in the Figure), which holds a reference to the top scope leader (or the master node, used in the case studies of the evaluation chapter 7). When created, an instance of the Scope model element represents the scope leader of that scope.

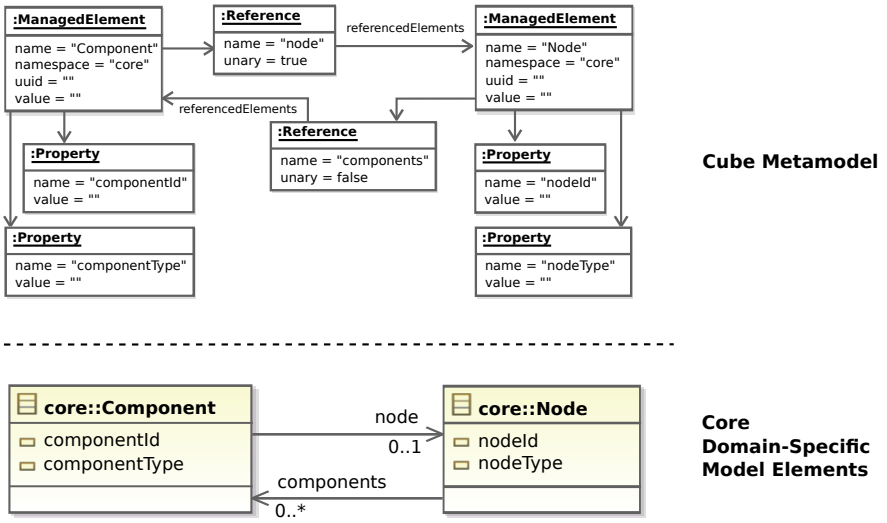


Figure 5.7: Core Domain-Specific Model fragment and its corresponding Cube Meta-model instance

These core managed elements specific to the mediation domain are defined using the Cube Meta-model. Figure 5.7 details an example of the relationship between a part of the core domain model and its corresponding Cube Meta-model instances. As before, this example shows the advantage that the domain-specific modelling layer (M1) brings with respect to the higher Cube meta-model level (M2), in terms of increased simplicity of model specifications.

5.2.4 Cube Runtime Model

Cube Runtime Model is located at the bottom layer of our modelling stack (M0). At this level, we find instances representing the real elements of the managed system. We refer to these instances as Runtime Managed Instances, or modelled element instances. As we will show later (section 5.4.3), Cube framework provides support for a set of technology-specific plug-ins, or extensions, that monitor the managed system and create corresponding modelled element instances in the runtime model. These are instances of the domain-specific elements modelled before (in M1).

Modelled element instances are identified by unique identifiers – UUID. UUID (Universal Unique IDentifier) is an identifier standard coded in 128 bits and generated using a pseudo-random algorithm that takes into account the computer characteristics (e.g. hard drive serial number and MAC address). It guarantees uniqueness across space and time. UUID were originally used in the Apollo Network Computing System and later in the Open Software Foundation's (OSF) Distributed Computing Environment (DCE) [Sch93], and then in Microsoft Windows platforms (known as GUID – Global Unique Identifier) to refer to software components. In the Cube context, UUIDs enable distributed Cube AMs to uniquely identify Runtime Managed Instances without relying on central coordination.

At runtime, each Cube Autonomic Manager administers a part of the managed system. Since we adopted a model-based management approach (subsection 5.1.2) each Autonomic Manager maintains a runtime model of the system part that it manages. Hence, this actually represents a runtime model part of the entire runtime model of the entire managed system. In its runtime model part, an Autonomic Manager maintains a set of Runtime Model Instances that represent the managed system elements which it administers. All the modelled instances are represented based on a coherent format, which conforms to the Cube Meta-model. This makes the runtime model easy to inspect via automatic tools, either for displaying state information to human administrators or for allowing an Autonomic Manager to analyse it. A runtime model can be viewed as a graph where the vertices are the Runtime Model Instances and the edges are the references represented by UUID identifiers. Hence, an automatic function can easily navigate through the runtime model, inspecting each Runtime Model Instance, analysing its properties and following its references to find related Runtime Model Instances.

Figure 5.8 shows an example of two associated Runtime Model Instances – Component instances in this example – located on different nodes and hence managed by two different Cube AMs. Most importantly in this figure is to notice how the association between the two component instances is represented. The first component instance that has the component type "`Encoder`" and component identifier "`Encoder-1`" is hosted and managed by a Cube Autonomic Manager that has the following URI: "`cube://192.168.0.1:38`". This component instance has the generated UUID "11OE-8400-1234-9876" (shortened for clarity). It also has a reference named "outputs", which represents its output components. This reference indicates that the component instance has one reference to another component instance, which is identified by its UUID: "`34E1-763E-1298-9988`". This UUID identifies another component instance, which is located under the administration of another Cube Autonomic Manager, with the URI "`cube://192.168.0.2:38`". This second component instance has the component type "Decoder" and the component identifier "`Decoder-1`". Its UUID corresponds to the one referenced by the initial component instance.

Finally, we notice that Runtime Managed Instances have no functional code; they are just modelling abstractions for representing the managed elements handled by the Cube framework. Each Runtime Managed Instance contains a list of its managed properties and its references to other Runtime Managed Instances representing other managed elements.

More information about the Cube Runtime Model will be provided later (section 5.4.2).

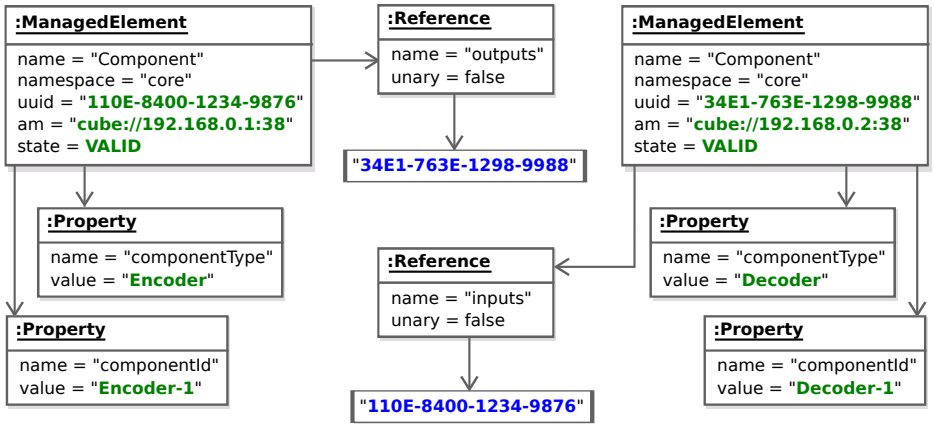


Figure 5.8: Cube Runtime Model Example

5.3 Goal Description Language - GDL

To automate management tasks for software systems, we need a formal language for specifying administration goals with precise, but simple syntax. In this thesis, we propose the *Goal Description Language* (GDL) for specifying objectives within the Cube framework. It is the formal language for defining the Cube archetype.

GDL enables system administrators to specify management objectives as constraints on managed system elements. Here, the managed elements modelled previously via the SML (subsection 5.2) can be used as parameters to GDL constraints. The resulting specification – a Cube archetype – is provided as input to all Cube Autonomic Managers that participate to the system's administration. This input specification represents the administration goals that the Autonomic Managers must attain.

In the following, we will introduce the concepts and syntax of GDL. For better clarity, the GDL syntax is abstracted and represented here via graphical directed graphs. The actual XML syntax of GDL is described in the next Chapter – "Implementation".

At its core, GDL is a formal model for describing the administrative goals that must be attained on a set of managed elements. Hence, GDL can describe the managed elements targeted for self-management and specify the goals to be achieved on these elements. Goals can be expressed in terms of properties to be attained on one managed element, or in terms of references to be ensured between two managed elements. Examples include two components of a particular type that must always be connected; or a server's maximum number of clients accepted. Certainly, the manner in which such goals are achieved can depend on the distributed execution environment.

The GDL syntax can be abstracted in the form of a simple graph-based model. It enables the simple representation of goals and statements about managed elements as a directed graph. Here, vertices represent the managed elements, and edges represent their targeted properties or the relations between them (e.g. references to other elements). The main objective of the GDL model is to allow administrators to specify self-management objectives in terms of the constraints that the system should meet during its execution. This objective translates into specifying sufficiently-detailed descriptions of managed elements and the properties or references to be satisfied by the Cube framework at any one time.

Figure 5.9 shows a graphical representation of a GDL model example. In this example, we want to guaranty that any component of type "Decoder" is connected to another component of type "MailSender-

Component", which must be on a node of type "MailServer". Hence, the goal of this specification is to ensure that these two components are always connected. The purpose of all the other details provided is to specify with precision which components are subject to this self-management objective. We will detail the GDL model and the exact syntax of its graph-like representation in the next sub-sub-sections (5.3.1 and 5.3.2).

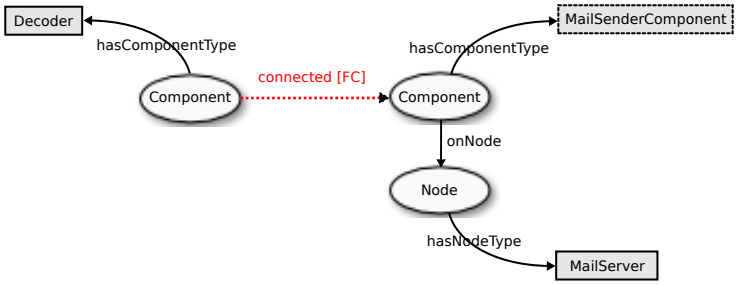


Figure 5.9: GDL Example

In general, GDL is based on the idea of making statements about managed elements in the form of subject-goal-object expressions. Here, the subject represents a precise description of a managed element, in terms of its properties, property values and references. The object is a description of another managed element, which is related to the subject element (via the goal). The goal expresses the characteristic(s) that the subject element must feature at any one time. In the GDL constraint exemplified above (Figure 5.9), the "Decoder" Component represents the subject, the "MailSenderComponent" the object and "connected" the goal – it indicates that any subject component must be connected to an object component.

If the *object* is a literal value, the goal is about controlling a property of the subject managed element (e.g. a value range for a component's attribute). If the object represents another managed element, the goal defines a binary property that must be ensured between the two elements (e.g. two components that must be connected). Notice that the expression subject-goal-object has also been used in other models like in the Resource Description Framework (RDF) [LS99], but based on another pattern – subject-predicate-object – and for another role – intended to describe Web Resources. The main difference is that RDF was intended to specify static web resources, in order to facilitate the automatic processing of their descriptions; while GDL is intended to specify user goals with rich descriptions about system architecture and constituent managed elements.

GDL is intended to be processed automatically by the Cube framework, rather than only to being displayed to human administrators. Furthermore, the model is based on an abstract architectural model rather than on the direct expression and description of managed system elements. Namely, vertices in the GDL graph are mere descriptions of the managed elements that should be part of the self-management goal (e.g. a Component of type "Decoder"). They are not concrete specifications referring to a unique managed element (e.g. not the full identifier of a particular Java class, or Component implementation from a particular provider).

The GDL language relies on three main concepts:

- Element (Managed or Primitive);
- Property (Goal or Description);
- Archetype.

Table 5.1 gives the graphical notation of these concepts (except Archetype which is a document in which the user write his specification).

Table 5.1: Graphical notation of GDL concepts

N°	GDL Concept	Graphical notation's example
1	Managed Element	Managed Element
2	Primitive Element	Primitive Value
3	Description Property (unary)	Managed Element Property name Primitive Value
4	Description Property (binary)	Managed Element Property name Managed Element
5	Goal Property	Managed Element Goal name [Resolution Strategy] Managed Element Property name Primitive Value Property name Primitive Value

In the following, we will detail each of these concepts of the GDL model.

5.3.1 GDL Elements

An Element is represented by a vertex in the GDL graph. It can represent two things:

1. **Managed Element** (ME): represents a description of a managed element, subject to self-management goal(s). It may represent any targeted managed element that has already been modelled based on the system modelling language (e.g. Component, Node and Scope).
2. **Primitive Element** (PE): represents a constant literal specified via a string value.

In the graph representation of GDL (Table 5.1), Managed Elements (ME) are represented as ellipses (Table 5.1-1), labelled with the ME's type; Primitive Elements (PE) are depicted as rectangles (Table 5.1-2), labelled with the PE's literal value.

5.3.2 GDL Properties

A Property in a GDL model can represent two things:

1. **Goal** (G): property of the system declared as an objective to achieve.
2. **Description** (D): provides more information about the subject or the object of a Goal.

In both cases, a property is triplet of the form: subject-property-object. This is a generalisation of the GDL statement format presented above – subject-goal-object – considering that the Property can actually be a goal (G) or a description (D). Furthermore, we can now discuss in more detail the nature of the subject and object elements based on the previous GDL Element definitions – Managed Element (ME) and Primitive Element (PE).

The subject is the element concerned by the property. Hence, it must be a Managed Element (ME) and cannot be a Primitive Element (PE). The object can be either an ME or a PE depending on the Property's cardinality, which can be unary or binary; n-ary properties are not supported at this point. More precisely, the object of a unary property is a Primitive Element; and the object of a binary property is a Managed Element. From another perspective, the object can be viewed as a parameter of the subject's property.

In GDL's graph-based representation, properties are depicted as edges (or arcs) and labelled with the property's name. Edges are directed from the subject to the object, in other words, from the element concerned by the property to the element providing more information about the property. Finally, Description properties are depicted via solid black edges (Table 5.1-3/4) whereas Goal properties via dotted red edges (Table 5.1-5).

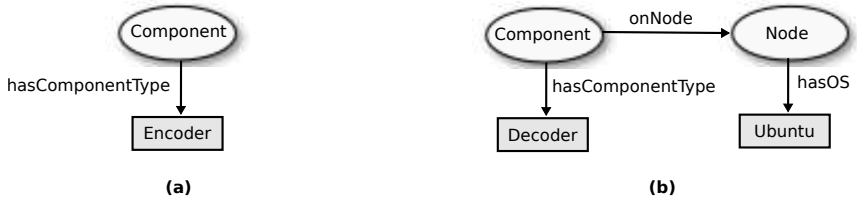


Figure 5.10: Examples of Description Properties in the GDL model

Figure 5.10 illustrates two examples of GDL properties used as Descriptions (solid dark edges). Figure 5.10-a depicts a unary property. The Managed Element of type "Component" – the subject – has a unary property "hasComponentType", where the object is a Primitive Element assigned the string value "Encoder". This simple Description property is interpreted as follows: "any Component that has the type Encoder". Figure 5.10-b depicts a binary property. Here, a Component is described as having the type "Decoder" and also as being deployed on a Node that has an "Ubuntu" Operating System (OS). This second GDL specification can be interpreted as follows: "any Component that has the type Decoder, and that is on a Node that has an Ubuntu operating system". At this point, no goal has yet been specified. These are just Descriptions that can be used later on for defining the actual goals (Goal Properties).

When GDL properties are used as Goals (dotted, red arcs), they represent the objective to be achieved. In Figure 5.11, the property connected is specified to be a goal to achieve. That is, the Component that has the type "Encoder" must be connected to another Component of type "Decoder", which is on a node that has an "Ubuntu" OS.

Compared to Description Properties, Goal Properties provide additional information about the Autonomic Manager's Resolution Strategy. This information is useful for optimizing the resolution process. In Cube, when trying to achieve user goals, an Autonomic Manager uses an Archetype Resolver – an internal module which is detailed later (sub-sub-section 5.4.3). The important aspect here is that the Archetype Resolver builds an internal resolution graph, based on the archetype, and tries to find runtime model configurations that satisfy the goal properties and descriptions in the graph. Considering the example depicted in Figure 5.11, the Archetype Resolver tries to achieve the goal connected for the subject, which is a Component of type "Encoder". To achieve this goal, the Archetype Resolver first tries to find the goal's object, which is a Component of type "Decoder" and is located on a node that has an "Ubuntu" OS. The search

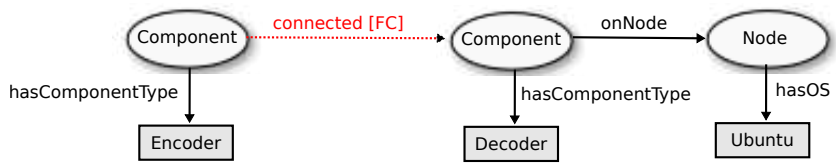


Figure 5.11: Example of a Goal Property in the GDL model

is carried-out based on the information available in the Runtime Model. If no instance of a component that conforms to this description if found, the Archetype Resolver creates a modelled component instance with this description and places it in the Runtime Model. This find (F) or create (C) behaviour is explicitly specified in the connected goal via the literals (FC).

The Archetype Resolver developed as part of Cube framework's prototype supports the following Resolution Strategies:

- **FIND** [F]: find the modelled element instance that conforms to the description provided by the object element of the binary property.
- **FIND_OR_CREATE** [FC]: start by looking for a modelled element instance that has the given description; then, if not found, create this instance in the runtime model.
- **CREATE** [C]: directly create an instance with the description provided in the object element without trying to look for an existing instance.

5.3.3 Archetype

The Archetype is the document where the GDL description is specified for each managed system. Formally speaking, an Archetype is defined as follows:

$$Archetype = \{E, P, G\} \tag{5.1}$$

$$\text{where} \begin{cases} E = \{e \mid e \ \ is\, a\, GDL\, Element\} \\ P = \{p \mid p \ \ is\, a\, GDL\, Property \wedge SubjectElement(p) \in E \wedge ObjectElement(p) \in E\} \\ G = \{g \mid g \in P \wedge g \ is \ a \ GDL\, Goal\, Property \wedge ResolutionStrategy(g) \in \{F, FC, C\}\} \end{cases}$$

- $SubjectElement(p)$ is a function that returns the Subject Element of a GDL Property p.
- $ObjectElement(p)$ is a function that returns the Object Element of a GDL Property p.
- $ResolutionStrategy$ defines the resolution strategy that the Cube's Archetype Resolver will follow in order to achieve the specified goal.

In addition, we provide the following formal definitions of the different parts of the Archetype:

$$E = ME \cup PE \tag{5.2}$$

$$\text{where} \begin{cases} ME = \{m \mid m \in E \wedge Type(m) \in \{Domain\,Specific\,Modelled\,Elements\,(M1)\}\} \\ PE = \{p \mid p \in E \wedge Type(p) = String\} \end{cases}$$

- ME is the set of GDL Managed Elements that have types defined in Cube's Domain-Specific Modelling Layer (M1 in Figure 5.2): Component, Node, Scope, and so on.
- PE is the set of GDL Primitive Elements which have the String type and have a given value.

$$P = G \cup D \tag{5.3}$$

$$\text{where} \begin{cases} G & already\,defined\,in\,(1.1) \\ D = \{d \mid p \in E \wedge d \;\; is\,a\,GDL\,Description\,Property\} \end{cases}$$

This formula indicates that GDL Properties can be either GDL Description Properties or GDL Goal Properties.

$$\forall\, p \in P \,:\, SubjectElement(p) \in ME \wedge SubjectElement(p) \notin PE \tag{5.4}$$

This means that the Subject Element of each Property (Goal or Description) is a Managed Element and not a Primitive Element.

$$\forall\, p \in P \,:\, if\;\; ObjectElement(p) \in ME \;\; then\;\; p\;\; is\,a\,Binary\,Constraint \tag{5.5}$$

This means that if the Object Element of a Property is a Managed Element, then p is said to be a Binary Constraint. It relates two Managed Elements together by a description or by a goal to achieve.

$$\forall\, p \in P \,:\, if\;\; ObjectElement(p) \in PE \;\; then\;\; p\;\; is\,a\,Unary\,Constraint \tag{5.6}$$

This means that if the Object Element of a Property is a Primitive Element (i.e. a literal value), then p is said to be a Unary Constraint.

Note: In the remainder of this thesis, when speaking about GDL graphs we refer to the Archetype which represents the same thing. However, there should be no confusion between the Archetype, which means the GDL graph, and the overall Goal and Knowledge Specification, which encompasses both Cube's System Modelling Layers and Languages SML (subsection 5.2) and the Goal Description Language GDL (subsection 5.3).

5.4 Cube Autonomic Manager

At runtime, a system's administration solution based on Cube framework consists of a set of collaborating Autonomic Managers that share the same goals. The goals are provided in the form of an Archetype document (presented above) and used by Cube's Autonomic Managers (AMs) to carry-out self-management operations. In this section, we detail the internal architecture of one Cube Autonomic Manager (AM). We present its core internal modules, as well as its various extension points and the relationships between them. We also show how a Cube Autonomic Manager uses the Archetype to resolve system management problems.

5.4.1 Internal Architecture Overview

In general, the design of the Cube Autonomic Manager (AM) follows the rather 'classic' autonomic computing MAPE-K[5] loop proposed by IBM [KC03]. In addition, it also proposes a few important complements. Figure 5.12 shows the internal architecture of an Autonomic Manager as defined in the Cube framework.

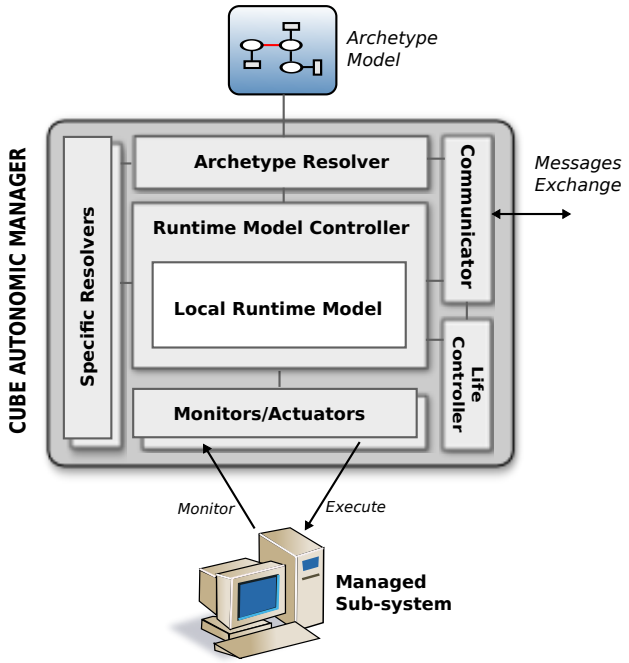


Figure 5.12: Cube framework's Autonomic Manager

Monitoring and execution modules in a Cube AM have the same role as in the MAPE-K loop. Similarly, planning and analysis tasks in the MAPE-K loop are equivalent to Cube's Archetype Resolver module. The Archetype Resolver provides the decision-making function of a Cube AM. It reasons on the system Knowledge – represented in the Local Runtime Model - and pursues the goals defined in the Archetype model. The Local Runtime Model represents the AM's partial view of the managed system, described via modelled element instances (or simply modelled instances) and their interrelations. All modelled instances of all Cube AMs constitute the Global Runtime Model, which reflects the overall state

[5]MAPE-K: Monitoring, Analysis, Planning, Execution – Knowledge.

of the entire managed system. Access to the Local Runtime Model, for reading and writing purposes, is mediated by a Runtime Model Controller via a special-purpose access interface. The Archetype model represents the user goals to achieve. As previously discussed (subsection 5.3), the AM compares the system's state, as indicated in the Local Runtime Model, with the Archetype's constraints and aims to maintain the system in states that conform to the archetype. This procedure is only performed on the part of the managed system that the AM administers locally.

When a goal cannot be achieved locally, a Cube AM must collaborate with other Cube AMs. For this purpose, it relies on an internal Communication module to exchange asynchronous messages with the AMs that it collaborates with – called "neighbours". Once related, a Cube AM also verifies the well-being of its neighbour AMs by using a special-purpose internal module – a Life Controller (or a Failure Detector). The purpose of this module is to verify the persistence of goal solutions that were set in place in collaboration with neighbours – e.g. remote element instances that help solve a connect constraint may disappear when a neighbour AM breaks-down.

To find solutions to system-specific management goals The Archetype Resolver makes use of Goal Resolvers. This enables the Cube AM to support the inclusion of new management concepts, new managed elements and new constraint resolution functions, without changing its core internal architecture.

To ensure the extensibility necessary for supporting future business objectives and various types of managed resources, the internal architecture of the Cube AM features two kinds of modules: core modules and extension modules.

Core modules are implemented and provided by the Cube framework. They include the Runtime Model and its special-purpose Controller, and the Archetype Resolver.

Extension modules can be added to the Cube AM for enriching its functions with system-specific capabilities. This includes the following modules: the Communicator, Monitors/Executors, Life Controller and Goal Resolvers. Extension modules allow specializing Cube's AM for specific management cases and execution environments. All the extension modules can be dynamically changed, added and removed as necessary.

In the following sections, we detail each of the internal modules of the Autonomic Manager in the Cube Framework.

5.4.2 Runtime Model Controller

The Runtime Model Controller (RMC) manages access to the modelled element instances available in the Local Runtime Model. The Local Runtime Model corresponds to the local knowledge that the Cube AM uses to take local decisions. It represents an abstraction and a direct representation of the managed system's state. More precisely, since it is a local model, it only represents the state of the managed system part that the local AM administers.

The Runtime Model Controller plays two roles. First, it is a container for the Local Runtime Model. This signifies that the Runtime Model Controller encapsulates and manages all access to the instances of modelled system elements available in the Local Runtime Model (e.g. modelled component instances and interconnections). The Controller also provides a notification mechanism to inform other modules about changes in the Local Runtime Model.

Second, the Runtime Model Controller provides a transparent API for distributed operations that can be used to retrieve information about remote modelled instances - that is, modelled instances that are located in remote Runtime Models and managed by other Cube AMs. In the following two sub-sections, we detail these two roles.

Local Runtime Model

A Local Runtime Model consists of a set of modelled instances, which belong to the Runtime Model Layer (M0) of Cube framework's System Modelling Layers (subsection 5.2.4).

Formally speaking, the Runtime Model is defined as follow:

$$RuntimeModel = \{V, I, U\} \tag{5.7}$$

$$\text{where} \begin{cases} V = \{\, i \,|\, i \;\; is\, Modelled\, Element\, Instance \;\; \wedge \;\; state(i) = VALID \,\} \\ I = \{\, i \,|\, i \;\; is\, Modelled\, Element\, Instance \;\; \wedge \;\; state(i) = INVALID \,\} \\ U = \{\, i \,|\, i \;\; is\, Modelled\, Element\, Instance \;\; \wedge \;\; state(i) = UNMANAGED \,\} \end{cases}$$

- $Runtime\ Model\ Instance$ is an object of the class `ManagedElement` defined in the Cube Metamodel (M2) (see 5.2.2).
- $state(i)$ is a function that returns the state of the Modelled Element Instance i. The possible states are `VALID`, `INVALID` and `UNMANAGED`, as discussed below.

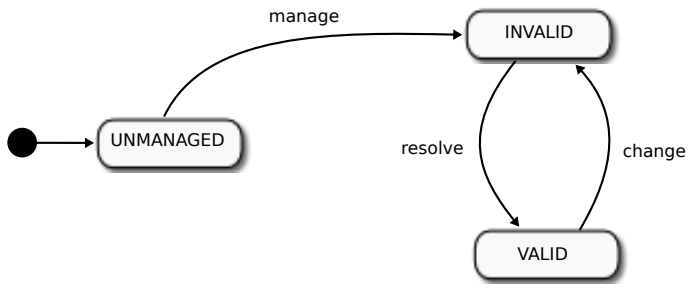


Figure 5.13: State Diagram of a Modelled Element Instance

The Runtime Model Controller changes the state of a modelled element instance (or modelled instance) according to its validity. Figure 5.13 shows the different possible states. When a modelled instance is created initially, it is tagged as `UNMANAGED`. This means that the Cube AM does not handle this modelled instance and does not try to validate it against the Archetype model. Unmanaged instances are useful for off-line manipulation. By off-line, we mean that no listeners are subscribed to notifications on the modelled instance's internal changes. Unmanaged instances are particularly used by the Archetype Resolver for checking their viability off-line, without causing any real impact on the managed system. Namely, the Resolver can create unmanaged instances and analyse their properties without activating the AM's resolution process.

When an AM takes a modelled instance into account - i.e., manages it - the Runtime Model Controller changes the instance's state to `INVALID`. The modelled instance remains in this state while it is not yet resolved and validated against the Archetype model; when validated its state is changed to `VALID`. At this point, listeners that have subscribed to the Runtime Model are notified. When properties of the modelled instance or of its references change, and if these properties are subject to goals in the Archetype, then the modelled instance becomes `INVALID`. The Archetype Resolver tries to validate it again (as described after in 5.4.3).

Global Runtime Model

Cube does not create or maintain any central Global Runtime Model of the entire managed system. As we have seen, the Runtime Model is partitioned amongst Cube's decentralised AMs. However, if necessary, we can retrieve a snapshot of the Global Runtime Model and navigate through it by following the local and external references (UUIDs) of modelled instances.

For this purpose, the Runtime Model Controller provides a distributed API that allows retrieving information about `VALID` remote modelled instances, which are hosted and managed by other Cube AMs. This facility is very helpful for developers especially when implementing Specific Resolvers (see 5.4.3). The API provided by the Runtime Model Controller includes the following methods:

- `getPropertyValue`: gets a property value of a modelled instance identified by its UUID. Property's name should be provided as parameter along with the UUID of the instance.
- `addProperty`: adds a property to the modelled instance identified by an UUID given as a parameter. Property's name and initial value should be provided as parameter.
- `updateProperty`: updates the value of a property of a modelled instance identified by an UUID given as a parameter. Property's name should be provided as parameter to identify which property should be modified.
- `getReferencedElements`: gets the list of the referenced modelled element instances (the list of their UUIDs).
- `addReferencedElement`: adds a new reference to the identified modelled instance identified by it UUID given as parameter.
- `removeReferencedElement`: removes the reference of a given element from the list of referenced element of another element.
- `hasReferencedElement`: checks is a given element is among the referenced elements of another modelled element instance.

These methods are all synchronous, meaning in this context that callers are blocked between the moment when they make the method call and the moment when they receive the response, or when a time-out exception is raised.

5.4.3 Archetype Resolver

The *Archetype Resolver* (or simply *Resolver*) (Figure 5.14) is the core component of the Cube Autonomic Manager (AM). Its role is to validate the Local Runtime Model against Archetype constraints and to modify modelled instances as necessary to bring and maintain the system within its viable solution space. Hence, the main task of the Archetype Resolver consists in a search algorithm that finds valid solutions starting from the monitored system state as represented in the local model.

There is no central Resolver in the Cube framework. Rather, each AM contains its local Resolver that verifies and updates the Local Runtime Model corresponding to the AM's managed system part. An AM's Resolver also collaborates with Resolvers of neighbour AMs in order to find more global solutions (see further in this section).

An Archetype Resolver provides a general-purpose, graph-based search algorithm for finding solutions. Nevertheless, in order to do so it must rely on user-provided Specific Resolvers associated to domain-specific modelled elements. In the remainder of this section, we detail the main algorithm proposed to resolve Archetype goals. We show how it is associated to Specific Resolvers and to domain-specific modelled elements. In particular, we insist on the "Scope" and "Node" concepts which are likely

to be mandatory for the algorithm's distributed version.

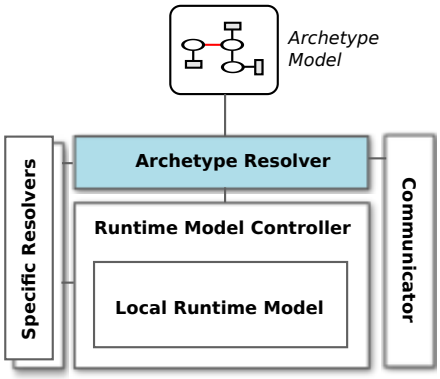


Figure 5.14: Cube AM's Archetype Resolver

The Archetype Resolver explores the system's solution space, which contains all valid system states that conform to the Archetype. This is done by incrementally modifying the current model (which reflects the system state) and checking it against archetype constraints. The Resolver stops as soon as a valid solution is found and set in place. Taking a simple example, let us assume that the Archetype imposes that a component A is always connected to a component B. When an instance of component A is created, the Resolver must find an instance of component B and connect it to A, without breaking other constraints related to A or B (or to any other types). The Resolver can find the B instance either in the Local Runtime Model, or in a remote Runtime Model. The manner in which the Resolver searches for solutions depends on the details that the user associates to goal definitions in the Archetype.

There are two concerns that the Archetype Resolver should take into consideration for ensuring user objectives as specified in the Archetype: representation and reasoning. Representation concerns the manner in which the goals are formulated, or modelled, and whether they are generic or application-specific. Reasoning refers to the manner in which solutions are identified, via search or inference-based procedures, so as to satisfy user goals. In our thesis, we have adopted the concept of Constraint Satisfaction Problem (CSP) as a solution for representing and reasoning about user goals. In the following, we start by introducing the CSP concepts. Then we show how the Archetype concepts are mapped to CSP concepts in order to obtain a representation of the problem model. Finally, we detail our proposed reasoning algorithm, based on an extensible architecture that allows the integration of user-specific resolvers, specific to each domain.

Definitions

"Constraint programming is a powerful paradigm for solving combinatorial search problems that draws on a wide range of techniques from artificial intelligence, computer science, databases, programming languages, and operations research" [RvBW06]. It promotes an approach where only the properties of the desirable solutions are specified, in the form of constraints or relations between variables. The constraint-oriented program must then find solution(s) that conform to the specified constraints. In contrast, imperative programming specifies directly the exact set of instructions to be executed on a set of variables in order to find a solution.

Variables in constraint programming can have Boolean, integer, linear, or finite values. Finite domains are one of the successful areas where constraint programming is applied [RvBW06]. They include

scheduling, planning, vehicle routing, configuration, networks and bioinformatics domains. Such real-life problems deal with multiple finite alternatives and so can be represented as Constraint Satisfaction Problems (CSP) and resolved by applying various Constraint Programming tools. Constraint satisfaction, in its basic form, involves finding a value to each variable in a problem's variable set, where constraints specify that some subsets of values can(not) be used together.

Formally speaking, a CSP is defined by a triplet - $< X, D, C >$ - where X is the set of variables, $X_1, X_2, ..., X_n$; C is a set of constraints, $C_1, C_2, ..., C_m$; and D is the set of non-empty domains, $D_1, D_2, ..., D_n$, where D_i defines the set of possible values of variable X_i. Each constraint C_i involves a subset of the variables and specifies the allowable combinations of values for that subset. A state of the problem is defined by an assignment of values to some (or to all) of the variables, $\{X_i = v_i,\ X_j = v_j,\ ...\}$. An assignment (or state) that does not violate any constraints is called a consistent or legal assignment. A complete assignment is one in which every variable is mentioned, and a solution to a CSP is a complete assignment that satisfies all the constraints.

$$
\begin{array}{rcl}
X & = & \{x_1,\ x_2,\ x_3\} \\
D(X_i) & = & [0, 2],\ \forall\, x_i \in X \\
C & = & \begin{cases} c_1 & : \ x1 < x_2 \\ c_2 & : \ x1 + x_2 \geq 2 \\ c_3 & : \ x1 < x_3 \end{cases}
\end{array}
$$

Figure 5.15: Example of a Constraint Satisfaction Problem (CSP)

Figure 5.15 presents an example of Constraint Satisfaction Problem (CSP). We aim to find values for x_1, x_2, and x_3 knowing that these variables can take only integer values between 0 and 2 and are respecting the three mentioned constraints.

The Resolution Graph - mapping of archetype model to CSP model

A constraint on a single variable is called unary constraint; and on two variables binary constraint. When all CSP constraints are unary or binary, the CSP can be directly represented as a constraints graph, with variables as vertices and constraints as edges. This is equivalent to the Archetype model (GDL graph) with Managed Elements representing vertices and properties representing edges. Hence, there is a direct mapping between the CSP graph and the Archetype graph.

Therefore, within the Cube framework, the Archetype Resolver constructs an internal constraints graph called *Resolution Graph* (RG) based on the Archetype model. This is the equivalent of a CSP graph. Cube's resolution algorithm (detailed further) uses this Resolution Graph (RG) to find viable solutions.

In a centralized control context, the CSP would be represented as one large constraint graph to be solved by a single resolver. This would mean that a centralised Archetype Resolver builds a Resolution Graph for the entire Archetype and then solves it. Let us now see how Cube's decentralised resolution process handles this problem instead. At runtime, Cube AMs maintain locally different parts of the Global Runtime Model. That is, each AM maintains a Local Runtime Model which it must validate against the corresponding part of the Archetype. This means that the AM only needs to construct a partial Resolution Graph that represents that part of the Archetype.

Here, using a graph structure to represent the Archetype's model as a CSP provides a key advantage. Namely, the Archetype's graph-based representation can be split into smaller sub-graphs, each one rep-

resenting a partial CSP, which an AM must resolve. Such sub-graphs can then be solved independently by the decentralised AMs and then merged into a global solution. Certainly, the applicability of this approach is limited to cases where the composition of partial solutions results in a global solution that conforms to the overall Archetype. This was the case in the data-mediation systems that we have studied as application examples and that represent useful and realistic applications in the targeted business domains.

Notice that we have designed the Archetype model to be a goal-oriented, architecture-based language. The resulting specification (detailed in section 5.3) is intended to be user-friendly, including little information related to the actual resolution process. We aim by this to make the Archetype model as independent as possible from the actual reasoning technique used to resolve it. One exception is the Resolution Strategy which is used to guide the Resolver's search process for component instances (e.g. find or create strategy); it represents a goal property and added as a tag to the goal definition.

In the Archetype, the user specifies a set of managed elements and their properties. Properties are defined either as attributes (*predicates*) or as relations between elements. This can be viewed as a set of *constraints* in the constraint programming paradigm [APS11].

Moreover, in Cube, the possible values for the set of variables (Xi) that belong to the corresponding domains (D_i) will change during runtime. For each variable of a certain type (e.g. component or node), the domain values that can be selected from consist of the modelled instances that are possible for that type (e.g. component instances that can be created or found; nodes that are available). The domain, which represents the solution space of a variable, can be further limited to more precise values using GDL Property Descriptions (subsection 5.3.2). Furthermore, the values can be located in the Local Runtime Model or in a remote Runtime Model. The search function is implemented and associated to GDL Descriptions. For instance, the on-node constraint receives the list of its associated components. If the on-node constraint is specified for a component, the possible values for that component's variable are limited to component instances that execute on the node indicated.

In the Resolution Graph, since a vertex represents a managed element (or variable), finding a legal value for that vertex is the equivalent of resolving a unary constraint. Similarly, since connections (edges) represent constraints between variables (vertices) selecting values for two connected vertices is the equivalent of resolving a binary constraint.

The Resolution Algorithm

The Archetype Resolver checks the Local Runtime Model for `INVALID` modelled instances and launches the resolution process to find a solution that can re-validate them. This is said to be a passive approach, where the Resolver executes this procedure repeatedly at a predefined interval. The Archetype Resolver also implements a reactive approach. In this case, when changes occur in the Local Runtime Model, the Archetype Resolver is notified automatically. It then checks if the new or updated modelled instance (which it was notified about) matches any Managed Element's Descriptions in the Archetype. If it does, then it starts to construct progressively and to solve a Resolution Graph that has the modelled instance as a root vertex and the associated constraints as directed paths departing from that root vertex. The Resolver continues to construct and solve the Resolution Graph locally (as described below) until it reaches the limits of its local capabilities. At this point, the Resolver may contact remote Resolver and hand over the resolution process. Collaborating Resolvers become neighbours and maintain contact during the entire resolution process (to check for alternative solutions) and after (to check for neighbour liveliness).

The resolution algorithm used by the Archetype Resolver is a distributed backtracking search algorithm based on a greedy find-and-test approach. To illustrate the algorithm's functioning in principle, let us consider the Resolution Graph representing the Archetype, and for each vertex (variable) let us consider all possible values (defined by the variable domain). Based on this view, the resolution algorithm

navigates through the space of possible solutions as follows. Starting from a given vertex (let us call it root vertex) it follows all possible paths in the directed graph starting from that vertex. In this process, it tries to find legal values for all vertexes in its paths. This is the equivalent of resolving the associated constraints. When this is achieved a solution is found. If this cannot be achieved then a solution could not be found.

More precisely, at each vertex, the algorithm selects one of the possible values, from the corresponding variable domain and considering the available system resources. This choice of value may constrain the set the values available in the connected vertices. If the algorithm can continue to follow the graph path and find values at each vertex until reaching the end of each path in the Resolution Graph, then a solution has been found. The Resolver enforces this solution onto the Runtime Model, which triggers its actual implementation into the managed system.

On the other hand, if the algorithm gets blocked (i.e., no values available at a vertex) then it goes back to the previous vertex and selects a different value from the list of possible values. If such value is available then the algorithm starts to go forward again through the Resolution Graph using the new value. If no alternative value is available at a vertex, the algorithm continues to go back on its path and try to select different values for vertexes increasingly close to the root vertex. If it reaches the root vertex and no further values are available then a solution cannot be found. In this case, the modelled instance that has fired the resolution process remains `INVALID`. The Archetype Resolver will try to resolve the `INVALID` instances again, either in the next round (passive approach) or when changes occur (active approach).

The resolution algorithm searches the solution space in a systematic manner that guarantees that all possibilities will be tried. It simply enumerates and tests all candidate solutions containing values for all variables in the Resolution Graph. When the Resolution Graph contains many vertices with large variable domains, the combinatory problem can become quite challenging and resource consuming. Nonetheless, the resolution algorithm stops as soon as a legal solution is found, rather than exploring all possible solutions and then selecting the best one. While this may lead to under-performing solutions, it also represents an advantage if a viable solution, even non-optimal, can be identified relatively fast (in comparison to the time necessary for a full exploration of the solutions space). Still, even in this context, we do not claim to provide an optimal resolution algorithm. This is simply a proof-of-concept that fits our architecture and specific needs. Several Artificial Intelligence (AI) techniques are available from the literature to improve the presented search algorithm – e.g., propagation, data-driven computation, "forward checking" and "look-ahead". Applying such optimizations is out of the scope of this thesis.

We notice finally that The proposed resolution algorithm is written in a generic way that applies to all administrative goals (core and future extensions) supported by the Cube framework.

5.4.4 Life Controller

The *Life Controller* (LC) verifies the liveliness of all the remote modelled instances which are referenced by modelled instances in the Local Runtime Model. It provides the verification and notification mechanism that is essential to Cube framework's self-healing facility. Namely, it verifies periodically that the different parts of the managed system are always correctly associated despite the dynamic nature of the managed system (e.g. devices that appear and disappear dynamically like in ubiquitous computing).

The Life Controller has a direct relation with the Local Runtime Model (see Figure 5.16). It is subscribed as listener and notified about changes in the runtime model. In particular, whenever a remote modelled instance is referenced from the Local Runtime Model, the Life Controller records the URI of the AM that administers the remote instance and starts checking on it (Algorithm 5.1).

This procedure is only activated for references that are subject to self-management goals specified in the Archetype document. An example is the connected goal which ensures that two components are

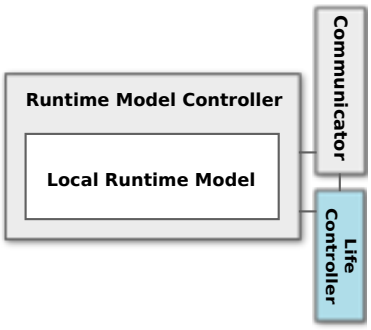


Figure 5.16: Cube framework's Life Controller

always be connected. If the two connected components are managed by two separate AMs (Figure 5.17), the Life Controller of the local AM, which administers the subject component, monitors the aliveness of the remote AM, which administers the object component.

Algorithm 5.1 Checking the aliveness of remote AMs (neighbours)

```
for each AM in MonitoredAMsList do
    send_aliveness_message(AM)
    if (aliveness[AM] < TENTATIVES) then
        aliveness[AM] = aliveness[AM] + 1
    else
        remove_references_of(AM)
end for
```

This ensures that if the remote AM goes down (e.g., the execution machine crashes), the local AM learns about it and its Archetype Resolver gets a chance to find another solution. More precisely, when receiving no response from a monitored AM, the Life Controller decreases the value of aliveness relative to that AM. When no response is received after a predefined number of attempts, this is interpreted as the remote AM being inactive (or broken).

At the same time, the Life Controller of the remote AM will be configured to notify the local AM whenever the object (remote) component is removed or changed. Notice that if no remote reference is found in the Local Runtime Model, the Life Controller does not maintain any monitoring channel (case of the `Cube AM 1` in Figure 5.17).

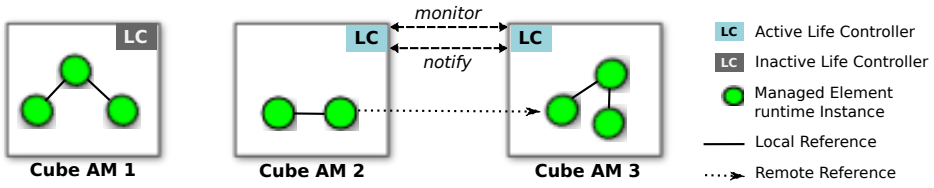


Figure 5.17: Example of Cube framework's Life Controller

To summarise the Life Controller's work, we highlight the following two tasks:

1. Notifying remote AMs about any changes of local modelled instances that are referenced by instances located on these remote AMs.

2. Monitoring remote AMs that host modelled instances which are referenced by local modelled instances.

For illustration purposes, the following algorithm (5.2) details the first task corresponding to the notification of external AMs managing external references when a local instance is changed. In this algorithm, we show the case of removing an instance. The Life Controller retrieves all its external references, and from its recorded URIs of the external AMs, it sends a notification message for each one that has a corresponding reference to one of its local instances.

Algorithm 5.2 Notify remote AMs about the removal of a referenced instance

```
function notify(i)
    for all (external reference of i) do
        am_uri = getAM(i)
        notify_removed_instance(am_uri, i)
    end for
end function
```

The `notify_removed_instance` function uses the Communicator module (see section 5.4.6) of the Cube AM (see Figure 5.16) to send messages to remote AMs. When the remote AM receives the notification message, it removes all occurrences of the remote modelled instance in its Local Runtime Model. Consequently, any modelled instances depending on the removed instance will be marked as `INVALID` as it was changed (see 5.4.2), and the Archetype Resolver will have to try to validate them again.

5.4.5 Monitors/Executors (Technology-Specific Extensions)

Most of the studied autonomic frameworks are often delivered with specific monitors, executors and control strategies. However, the modular and extensible architecture of Cube framework's AM allows the integration of user-specific modules in order to specialize the AM for handing technology-specific aspects.

In short, the control process proceeds as follows. Monitors collect data from technology-specific sensors and update the Local Runtime Model accordingly. The Archetype Resolver is notified of any such updates in the Local Runtime Model. It checks the conformity of the new Local Runtime Model to the archetype, takes the necessary decisions, and indicates any corrective changes by modifying the Local Runtime Model. Executors apply such changes into the corresponding managed elements, via technology-specific actuators.

Figure 5.18 shows an example involving technology-specific Monitors and Executors. Monitors create modelled element instances corresponding to the managed system components and their configurations. Executors apply changes made to the Runtime Model into the actual managed system. The exemplified Database system (on the left of the Figure) is monitored by a Cube Monitor implemented specifically for the corresponding Database technology. This Monitor creates an abstract modelled instance representing the Database and featuring its configuration information. The Database is connected to another part of the system, which is managed by another Cube AM (on the right of the Figure). Let us assume that this second system part relies on two different technologies, hence requiring two corresponding types of Monitors/Executors to handle modelled instances in the Local Runtime Model. Note that both of these technology-specific Monitors/Executors access the same Local Runtime Model and belong to the same Cube AM instance. The Runtime Model Controller mediates all access to the Local Runtime Model and ensures its coherence in the presence of multiple parallel updates.

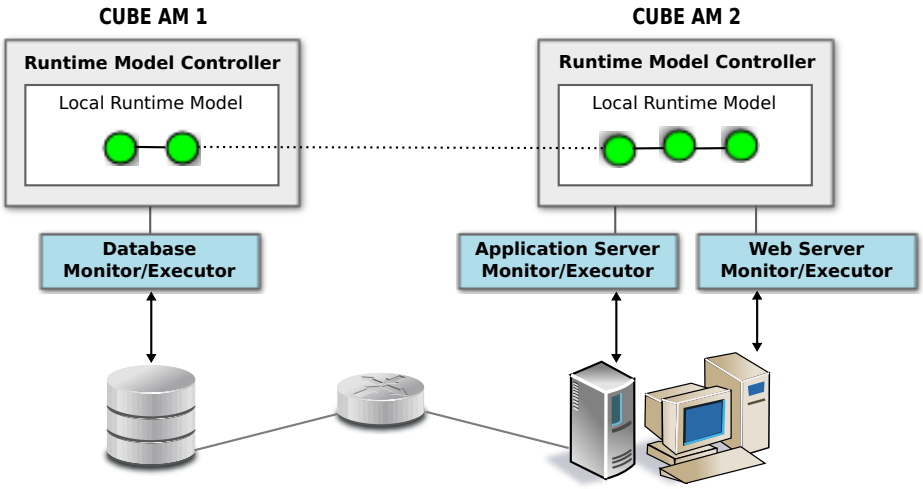


Figure 5.18: Cube Monitors/Executors

Technically speaking, the Monitor uses the Runtime Model Controller API to add and change modelled instances in the Local Runtime Model. The Archetype Resolver will validate the modelled instances that were created or updated by the Monitor. When such changes are validated, the Executor is notified and immediately applies these changes into the managed system. The Executor can set-in place filters for selecting the modelled instances from which it receives update notifications.

Monitors and Executors are initially loaded based on the way in which they are specified in the Cube AM's configuration. They can also be added or removed dynamically using a special-purpose administration API. Using this API (see next chapter) allows automating some tasks, such as starting or stopping dynamically a particular extension depending on the monitored context.

5.4.6 Communicator

Data exchanged between Cube AMs is mapped into a serializable Java Class called CMessage.

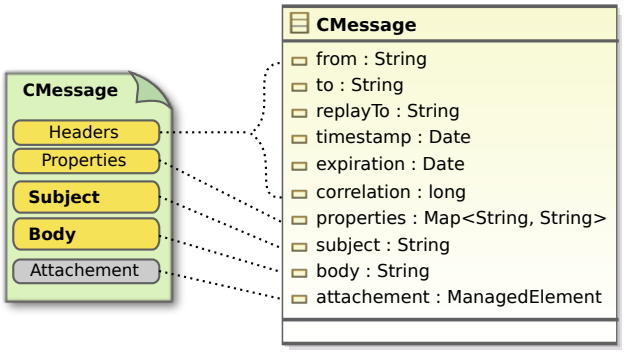


Figure 5.19: Cube Message Representation

CMessage consists of five main parts, as follows (see Figure 5.19):

- **Headers**: all messages support the same set of header fields. Header fields contain values used to identify and route messages. Some of them should be filled-in by the framework, including from,

timestamp, correlation id, and expiration time; while others are left to the message sender to fill-in, like the "to" header. The following list provides a short description of these headers:

- **From**: contains the URI of the Cube AM message source;
- **To**: contains the URI of the Cube AM message destination;
- **Replay-to**: contains the URI of the Cube AM to which the response should be returned;
- **Timestamp**: contains the date when the message was created;
- **Expiration-time**: contains the date when the message will expire (no longer be valid);
- **Correlation-Id**: holds a specific identifier object for linking one message with another. It typically links a reply message with its request message.

- **Properties**: holds application-specific properties, like key and value entries, that are transferred between Cube AMs. This is typically used by the Cube Resolver to send messages containing some properties to be found.
- **Subject**: holds the message subject. Typically, messages between Cube AMs are exchanged respecting some internal protocols. In the subject attribute, it is recommended to put the name of the task to be done.
- **Body**: it typically holds the task results to be forwarded to the requester.
- **Attachment**: It can hold a Managed Element object. This is used by the Cube Resolver to share Managed Element descriptions providing more detail than what can be stored within the "properties" attribute.

Message Delivery

Considering communication patterns such as "Point-to-Point" or "Publish/Subscribe", we can provide several protocol implementations for best addressing various deployment situations. All such customised implementations must implement the Communicator interface, which defines the two following methods:

- `send(CMessage)`
- `addListener(MessageListener)`

The send method is for asynchronous communication. Implementations or user plug-ins can also provide synchronized communication based on this asynchronous method (as detailed in the next chapter 6). The `addListener` method allows adding listeners for messages that are received.

Within the Cube framework prototype we have provided a default implementation of this communication interface. It uses Network Sockets to transfer the messages. For this default implementation and for any other implementation the Communicator should analyse the message header information, especially the "`to`" attribute, in order to know to which AM the actual message should be delivered. At that point, the message is serialized in a format defined by the transport protocol selected.

When a message is received, the message listeners are notified and a copy of the message is delivered to them. At this level, the Cube Autonomic Manager filters to whom the message should be delivered. When a listener subscribes for messages it should also provide the "`replyTo`" attribute since this is used by the Communicator to filter the received message. The listener can use the correlation header to ensure that the received message is what it was waiting for.

Details about the default implementation are provided in the next chapter 6 - "Implementation".

5.5 Summary

The goal and knowledge specifications were proposed in the Cube framework for allowing users to formally define administrative goals and model their managed systems. The proposed system modelling language (SML) defines four modelling layers with concepts ranging from generic to domain-specific and runtime elements. Each layer provides a formal language for the layer below and instantiates and extends concepts from the layer above. The most important layer defines Cube's Meta-model concepts, which enables users to model system managed elements and integrate them into the Cube framework. This layer is extended by a Domain-Specific Layer within witch domain experts can define a modelled elements specific to their domain. As part of the Cube framework prototype, we have defined such a domain-specific modelled elements for the targeted data-mediation domain.

The provided specifications also include a goal description language (GDL), which can be represented via a graph-based syntax. GDL's Archetype model enables users to specify precise goals for the managed system. More precisely, GDL statements (or constraints) take as parameters managed system elements that users have already modelled based on the domain-specific SML. In general, GDL constraints represent either properties to control and regulate on managed elements; or references to achieve and maintain between managed elements.

We have presented the internal architecture of a Cube Autonomic Manager and provided details on each of its internal modules. Most of these modules can be reused and extended so as to facilitate the adoption and customisation of the Cube framework across various targeted applications.

More information about the implementation details of the Cube framework is given in the next chapter.

Chapter 6

Implementation

Contents

6.1 Overview **104**
6.1.1 Supporting Technologies 104
6.1.2 Execution Platform 109
6.2 XML Syntax for the Archetype model **110**
6.2.1 The Archetype Document 110
6.2.2 Archetype Elements and Description Properties 111
6.2.3 Archetype Goals 113
6.2.4 Complete Example 114
6.2.5 Archetype Parser 115
6.3 Cube Execution Platform **117**
6.3.1 Overall Architecture 117
6.3.2 Cube Runtime and its Administration Service 117
6.3.3 Autonomic Manager 119
6.3.4 Runtime Model Controller 121
6.3.5 Data Structure and Implementation for the Archetype Resolver 124
6.4 Extensions **128**
6.4.1 Overview 128
6.4.2 Extension Points 129
6.4.3 Deploying and using extensions 134
6.4.4 Extensions provided by the Cube Framework 135
6.5 Related Tools **140**
6.5.1 Overview 140
6.5.2 Hot Deployer 140
6.5.3 Console Commands 141
6.5.4 Dynamic Web Monitoring Console 142
6.6 Summary **145**

In the previous chapter, we have detailed our proposed Cube framework. We have illustrated all its constituents and its underlying concepts and architecture.

In this chapter, we will detail the design and implementation of the proposed Cube framework, including its runtime infrastructure, extensions and utilities. We start this chapter by outlining the supporting technologies involved in the development of our framework. These technologies are also required for running Cube Autonomic Managers and for implementing extensions. In particular, our framework is based on two main technologies: OSGi [All07] and iPOJO [EHL07, EH07]. We provide an overview of the execution platform and the different dependencies between its main constituents. Each of these constituents is then detailed in separate sections of this chapter. In section 6.2, we provide information about the XML syntax we have proposed for our Archetype Model. This is followed by a description of the runtime platform in section 6.3, where we provide information about how we have implemented the different runtime entities. In section 6.4, we provide explanations about how to extend the core Cube framework in order to be able to handle new situations. Finally, we conclude this chapter by outlining the set of implemented tools that facilitate Cube's developer tasks (section 6.5).

6.1 Overview

6.1.1 Supporting Technologies

The Cube project provides offline support for developers to implement the various extensions and model elements, as necessary for setting in place a Cube-based solution (chapter xx). It also provides a runtime support that helps running Cube Autonomic Managers and provides support for dynamic adaptability of their internal modules. Our implementation of the Cube framework is based on light-weight Service-Oriented Architecture (SOA) - the OSGi Platform - and a Service-Oriented Component model - iPOJO. We detail these two complementary technologies and the reasons for their adoption in the following sections.

6.1.1.1 OSGi

OSGi [All07] – The Dynamic Module System for Java – is a specification for a service-based platform, which facilitates the componentization of software modules and applications and ensures remote management and interoperability of applications and services over a broad variety of devices. The OSGi specification has been proposed by the OSGi Alliance consortium[1].

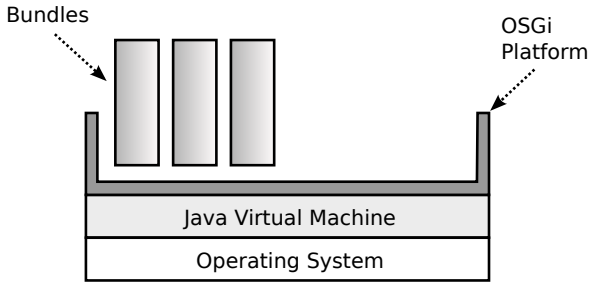


Figure 6.1: OSGi Platform

[1] http://www.osgi.org

The OSGi specification describes a Java execution platform that supports the dynamic deployment of modular and extensible applications. To be deployed onto an OSGi platform, an application must be packaged into a specific artefact called “bundle” (Figure 6.1). Bundles are JAR[2] files that contain the compiled classes and meta-information that defines the “OSGi bundle”.OSGi can run in several types of Java Virtual Machines (JVMs) ranging from very restricted virtual machines, like CLDC (Connected Limited Device Configuration) or CDC (Connected Device Configuration), to more desktop-based virtual machines, like the J2SE (Java Standard Edition). The OSGi platform is composed of several layers, from which the most important are the life cycle layer, the module layer, and the service layer (Figure 6.2). In next sub-sections, we detail these main layers.

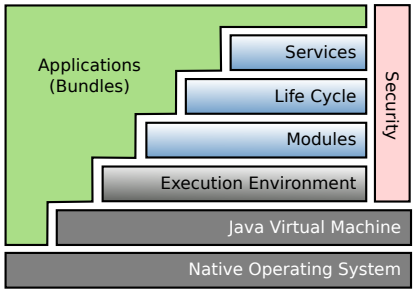


Figure 6.2: OSGi Layers

Module Layer

The Module layer is responsible for managing OSGi bundles that are deployed into the OSGi platform. It allows describing the sharing policies between different bundles, and hence it allows the construction of modular and dynamically updatable applications. Modules in OSGi are running bundles which contain meta-information about the sharing policies – the classes that the bundle exports, as well as the classes that it imports. This meta-information is exploited by the Module layer to automatically verify bundle compatibility and to interconnect bundles so as to resolve their required imports.

An existing Java application can be packaged (or "bundled") as an OSGi bundle by providing specific descriptors following the JAR specification. In general, an OSGi bundle must enrich the manifest with OSGi-specific meta-data that describe the module to be published into the OSGi platform, as shown in Figure 6.3-(a).

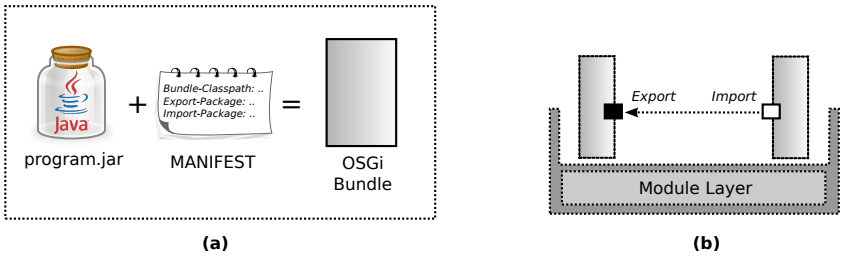


Figure 6.3: The Module Layer of the OSGi platform

[2]JAR (Java ARchive) is an archive file format used in Java platform to aggregate many classes, metadata, and resources.

The main OSGi-specific meta-data contained in the MANIFEST.MF artefact can be summarized as follows:

- *Exported-Packages*: the Java Packages that are provided by the bundle. Other OSGi bundles can reuse classes and services exported by this bundle (see 6.3-(b)).
- *Imported-Packages*: the Java Packages needed by this bundle. These Java Packages must be exported by another OSGi bundle available on the OSGi platform.
- *Activator*: the name of a special-purpose OSGi class – called Activator - which is responsible for managing the bundle (starts and stops it, as well as resolves and requiring services – see Lifecycle Layer next).
- *Bundle-ClassPath*: the list of the necessary Java Jars (inside the bundle) containing classes and resources needed to run the bundle.

Life Cycle Layer

Once an OSGi bundle is deployed onto the OSGi Platform, it will be managed by the OSGi Lifecycle layer, which will change the bundle state according to a set of specifications. This layer adds the possibility to install, uninstall, start, stop, and update bundles dynamically.

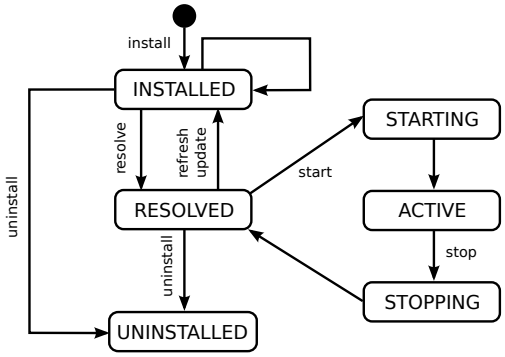


Figure 6.4: State diagram of an OSGi bundle's life cycle

Figure 6.4 shows the UML State Diagram for an OSGi bundle's life-cycle. When a bundle is deployed onto the OSGi platform, it is initially considered in the `INSTALLED` state; it becomes `ACTIVE` only when all its dependencies are resolved (i.e. when all its imported packages are available from other OSGi bundles installed on the platform).

Many OSGi implementations offer a shell via which the framework can be interacted with, using commands entered via a network connection or a local console. Currently, there is no OSGi standard on how this shell console should work, or how the console should be extended. However, different tools were proposed, like Gogo, Felix Web Console[3], and so on.

Service Layer

The interoperability of OSGi follows the simple Producer-Consumer paradigm of a service model as shown in Figure 6.5-(a). The Producer of an OSGi service registers with the OSGi platform's Service Registry, while the Consumer of the OSGi service uses OSGi's Service Look-up function to find and

[3]http://felix.apache.org/site/apache-felix-gogo.html

reuse the service. In this way, during an OSGi bundle's life-cycle, the provider can first publish their service implementation to the OSGi platform where other OSGi services (belonging to the same or to other OSGi bundles) can reuse it as show in the Figure 6.5-(b).

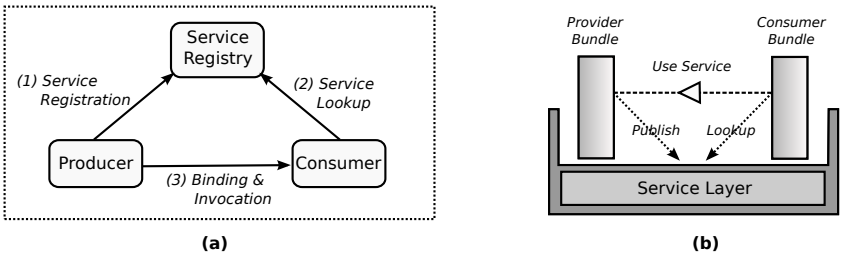


Figure 6.5: The Service Layer of the OSGi platform

The OSGi specification defines a dynamic Service-Oriented Architecture (SOA). This architecture has a centralized Service Registry, where services are described using Java interfaces and a set of properties whose semantics are left free to the users' choice. Service Producers register their service implementations in the Service Registry, and service Consumers look-up for particular services in the Service Registry. There is no direct link between service Producers and Consumers. Certainly, several Producers and Consumers can exist for the same Service. When a service Consumer finds a service Provider, a binding is performed between the two services.

These three OSGi layers - Module, Lifecycle and Service - allow the construction of dynamic and modular OSGi applications. However, when developing such applications it is essential to consider carefully the decoupling between bundles and to take into account the dynamism in the targeted Java code. Since this is a complex and repetitive task, special-purpose tools have been proposed to automate it, such as the Service Binder [CH03] and iPOJO [EHL07, EH07].

6.1.1.2 Apache Felix iPOJO

The OSGi technology provides a useful platform for creating dynamically-extensible and modular applications. However, dealing with dynamism and modularity can become cumbersome when the number of bundles and their dependencies increases. This is because OSGi developers must write a lot of extra code to handle such aspects and to manage all the related problems, such as thread synchronization.

Apache Felix iPOJO (acronym of *Injected Plain Old Java Object*) is a Service-Oriented Component (SOC) programming model built upon the OSGi platform. It was developed within the ADELE team[4] of LIG laboratory[5] – Grenoble. It combines the service oriented computing approach with component-based models to provide a programming and execution framework that allows developers to make dynamically adaptable components. It addresses most of the difficulties indicated above, by automating most of the repetitive tasks required by the OSGi platform.

From developer's point of view, iPOJO allows to hide the complexity of managing an application's dynamism. It simplifies the complexity of developing applications that will run on OSGi platforms. For this purpose, it automatically and transparently injects references between service providers and service consumers. It then manages such service dependencies transparently, during runtime.

Figure 6.6 shows the typical architecture of one iPOJO Component. It is composed of a POJO part (functional code) and a container (non-functional code). The POJO is a simple Java class that imple-

[4]http://adele.imag.fr
[5]http://www.liglab.fr

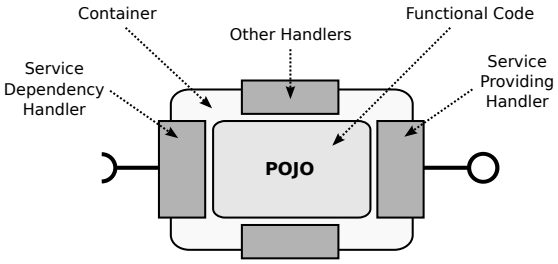


Figure 6.6: iPOJO Component-model

ments some application-specific function. It can implement a service interface, in which case the iPOJO component will provide an implementation of that service. It can also require a reference to a service implementing another service interface, in which case the iPOJO container will find a provider of that service and inject its address into the corresponding attribute of the POJO Java class. The iPOJO Component container is extensible; user-specific extensions (called Handlers) can be used to manage non-functional aspects related to the component (e.g., security, transactions and messaging). Service providing and service requesting are implemented as container handlers.

iPOJO proposes the following features:

- Tightly-coupled development model and execution platform: all the development models are available at runtime.
- Dynamic service-based architecture: the interaction between iPOJO components are issued from the underlying OSGi platform. In addition, iPOJO provides a certain services isolation. Developers can export services for their application's scope only.
- Simple development model: allows hiding the complexity of the underlying SOA using simple POJOs. Most importantly, it hides the dynamism complexity of the OSGi platform. iPOJO provides an execution layer with code injection and introspection features allowing for the transparent management of modularity and dynamism-related issues, as indicated via the developer's preferences.
- Structural composition language: iPOJO allows constructing applications from a structural composition of services. Composition is modelled based on service specifications, independently from concrete service implementations. This decoupling is an interesting feature of iPOJO as it allows selecting an available service implementation at runtime. Hence, applications built using iPOJO are dynamic.
- Introspection and dynamic reconfiguration: These mechanisms allow to reconfigure the iPOJO components and to retrieve their state during runtime.
- Extension mechanisms: iPOJO allows adding user-specific handlers of components to manage non-functional properties like persistence, security and quality of service.

Through these mechanisms, iPOJO simplifies the development of dynamic service-based applications. In addition, allowing developers to provide customised code for managing non-functional properties is another important facility. For this reasons we have chosen iPOJO as a basis for implementing our Cube framework prototype.

OSGi and iPOJO technologies are used to implement the runtime execution platform of the Cube framework. Furthermore, iPOJO should also be used by developers to implement Cube extensions. The use of these technologies is described through sections 6.3 "Cube Execution Platform" and 6.4 "Extensions".

6.1.2 Execution Platform

Cube framework is designed and implemented as an OSGi service, which is to be used by external tools to deploy and start Autonomic Managers. Autonomic Managers will then self-manage software systems according to the Archetype specification provided by the user.

The Runtime Execution Platform of Cube's self-management layer is composed of several OSGi platforms. These platforms are identical, each one executing on a different machine in the managed system. Figure 6.7 shows the most important Cube bundles and their relationships. It also depicts the component instances (created from these modules) that represent the Cube system running onto one OSGi platform.

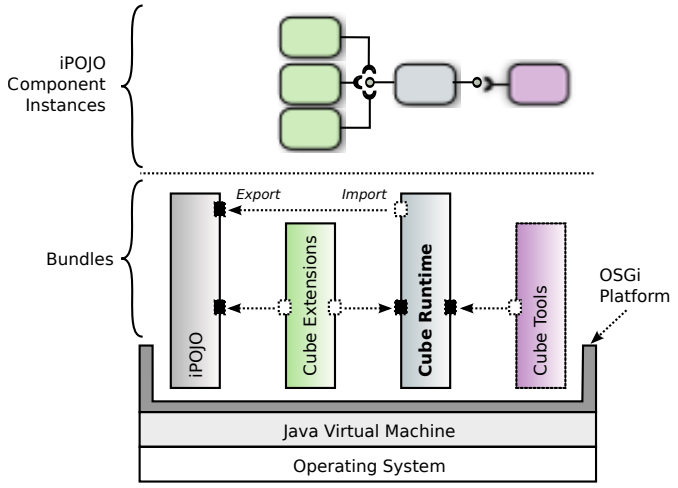


Figure 6.7: Cube's Execution Platform

More precisely, each OSGi platform has the following main bundles:

iPOJO Runtime

At runtime, Cube framework uses iPOJO to manage the different service-oriented mechanisms provided by OSGi. This includes mainly the management of required service references and service provisioning. It also allows the modularization of the internal architecture of Cube's Autonomic Manager and the seamless introduction of plug-in extensions. In our implemented prototype, we use the following iPOJO bundles:

- `org.apache.felix.ipojo-1.8.6.jar`: the core iPOJO runtime bundle.
- `org.apache.felix.ipojo.arch.gogo-1.0.1.jar` : OSGi commands for debugging and showing the exported factories and created instances.

Cube Runtime

A Cube Runtime is a software service that executes on a OSGi platform on each machine in the Cube system. Technically speaking, it exports an Administration Service that can be used to manipulate several Autonomic Managers running on the same OSGi platform. Cube Runtime is implemented as an iPOJO Component instance that exports an "Administration Service" which allows related tools to manipulate the Cube system, and requires an "Extension Factory Service" to allow the integration of new extensions to the running Autonomic Managers. Cube Runtime is detailed in section 6.3.

Cube Extensions

Extensions allow Cube's Autonomic Managers to administer domain-specific managed resources. Namely, they enable Autonomic Managers to interact with the managed resources by implementing adequate monitors and executors. Also, they enhance Cube's Runtime Model and Archetype Resolver with specific goals and their specific resolvers. Cube Extensions are detailed in section 6.4.

Cube Tools

Several tools can be implemented and deployed to provide specific tasks related to the functioning of the Cube Runtime. One such tools is the "Hot Deployer" which scans a particular folder, and, if it finds a configuration file for a Cube Autonomic Manager, it parses it and creates an Autonomic Manager based on the provided information. The implemented Tools related to the Cube framework are detailed in section 6.5.

When OSGi starts, instances of the aforementioned bundles are created and interconnected automatically with the help of iPOJO. We provide further details of each bundle and its functioning in sections 6.3 "Cube Execution Platform", 6.4 "Extensions", and 6.5 "Related Tools". Before that, we present the XML syntax for the Archetype Model in next section. This syntax allows system administrators to write formal Archetype specifications.

6.2 XML Syntax for the Archetype model

In the previous chapter "Cube framework", we have detailed the Archetype model used for describing user goals formally (see section 5.3 of chapter 5). To effectively use this model, in our prototype we have proposed an XML Syntax for the Archetype Model. Users write XML files (or generate them from a dedicated tool) and the resulting Archetype XML file is parsed by the Cube framework to construct its corresponding Archetype Java model (see section 6.2.5). This Java model is then used by the Cube Autonomic Manager as input to achieve the specified goals.

6.2.1 The Archetype Document

An Archetype is specified in an XML file, using a Cube-specific XML syntax. More precisely, an Archetype is defined as an XML element `<archetype>` which is a nested element of the root element `<cube>`. `<cube>` root element has only one attribute called cube-version which defines which version of the Cube implementation is intended to be used. At this level, we should also provide the list of XML namespaces that correspond to the Cube extensions to be included. As indicated previously, such extensions provide user-specific modelled system elements and their corresponding properties and goals. For illustrative purposes, the following listing 6.1 shows an example of an empty Archetype document.

Listing 6.1: Empty Archetype Document

```
<cube xmlns:core="fr.liglab.adele.cube.core" cube-version="2.0">
  <archetype
    id="net.debbabi.cube.demo"
    version="1.0"
    documentation="Empty Archetype">

  </archetype>
</cube>
```

The `<archetype>` element has the following attributes:

- **id** (compulsory): the identifier of the Archetype document. It is recommended to utilise a namespace-based identification (e.g., "`net.debbabi.cube.demo`").
- **version** (compulsory): the version of the Archetype document.
- **documentation** (optional): the description of the Archetype document. This will provide a global description of the goals intended by this archetype specification.

In the example in listing 6.1, we declared the "`fr.liglab.adele.cube.core`" namesapce which will be used subsequently to define user-specific managed elements (next sub-section). We also fixed Cube's implementation version to "`2.0`". Hence, this Archetype XML specification is compatible with version 2.0 of the Cube prototype. The declared Archetype document has "`net.debbabi.demo`" as identifier.

To support the archetype's evolution at runtime, it is important to have different Archetype identifiers for different Archetypes loaded by a Cube system. This means, that in principle, different Cube Autonomic Managers may load and pursue goals defined in different Archetype versions. In this manner, one given Archetype can have several versions that correspond to successive updates of the Archetype. Nonetheless, the current Cube framework does not yet support the dynamic update of the Archetype document. In the current version, in order to consider a new Archetype version, all the Autonomic Managers in a Cube system must be restarted. This limitation is detailed in the conclusion and perspectives chapter.

Finally, an Archetype XML element can also have an optional description which provides a short explanation of the role of this Archetype Document.

6.2.2 Archetype Elements and Description Properties

In the Goal Description Language (GDL) model - also referred to as Archetype model, see section 5.3 of chapter 5 - an Archetype consists of a set of GDL Elements, which have two declinations: Primitive Element (PE) and Managed Element (ME). PEs are primitive values - they are represented as attribute values in the Archetype XML Document. MEs are descriptions of managed elements - they are represented as XML elements in the Archetype. Each ME is defined via several Description Properties, which are defined as nested XML elements of the ME XML element. We detail the way of defining these types of elements next. The next subsection will show how Archetype Goals are defined with respect to such Managed Elements.

Each ME XML Element should be specified using a qualified name - i.e. using the extension's namespace and the Managed Element name. We recall here that Managed Elements correspond to the managed system resources that Cube has to manage, including Components and Nodes. An ME's corresponding XML element has the following attributes:

- **id** (compulsory): the ME's identifier.
- **documentation** (optional): the ME's documentation information.

The id attribute is obligatory, since it uniquely identifies the Managed Element within the Archetype XML Document. It can then be used and referenced from other XML elements, such as Goal or Description Properties (as discussed below). The description attribute is optional, since it provides an informal description of the Managed Element for the user.

Listing 6.2 shows an example of an Archetype with only one Managed Element of type Component. The Component concept is provided by the core extension of the Cube framework. This core extension is identified by the namespace `fr.liglab.adele.cube.core`. This is why this namespace was added to the list of namespaces of the Archetype XML file with a shortcut "core". To make use of the Component concept, we use the "`Component`" name appended to the extension namespace, so as to form a complete

qualified name that uniquely identifies the Managed Element type (in the example: `<core:component />`).

Listing 6.2: Example of an Archetype Element

```
<cube xmlns:core="fr.liglab.adele.cube.core" cube-version="2.0">
  <archetype   id="net.debbabi.cube.demo" version="1.0">
    <elements>
      <!-- list of elements -->
      <core:Component id="c1" />
        <!-- Description Properties go here -->
      </core:Component>
    </elements>
  </archetype>
</cube>
```

The declared Managed Element of type Component has "`c1`" as identifier. Based on the specification presented so far, `c1` now refers to any Managed Element of type Component. At this point, we have not provided any further precisions or details about the desired element, or component. This is the role of the Description Properties (DPs).

Description Properties are represented as nested XML elements of Archetype elements. A Description Property uses a fully qualified name and, when used as a child XML element of a Managed Element, it adds more details to the ME's description. This allows to better specify the exact kind of element that is being referred to, like a component or node featuring specific properties. Recall that a Description Property is a triplet of the form subject-property-object (section 5.3.2 - "GDL Properties"). In the Archetype, the subject element is the parent XML element of the Description Property's XML element. Hence, in the example above, the Component element (line 5) is the subject for all the Description Properties nested within it (line 6). The property is the qualified name of the Description Property's XML element (exemplified below). Finally, the object is represented as an attribute of the Description Property's XML element (exemplified below).

Overall, a Description Property element is defined using a qualified name (i.e. the DP's actual property), and can have the following attributes:

- **o**: the object of the Description Property;
- **documentation**: a description of the role of this property for the parent element (i.e. the DP's subject).

If the Description Property represents a unary property, then the object is a simple primitive value - the value of the '`o`' attribute. However, when defining a binary property, the object is another Managed Element. In this case, the value of the '`o`' attribute should start with a "`@`" character followed by the identifier of the Managed Element representing the object.

Listing 6.3 represents an extract of an Archetype document highlighting the use of Description Properties for providing more detail about targeted Managed Elements.

Listing 6.3: Example of an Archetype Element

```
<elements>
  <core:Node id="pc">
    <core:hasOS o="ubuntu"/>
  </core:Node>
  <core:Component id="c1" />
    <core:onNode o="@pc"/>
  </core:Component>
</elements>
```

In this example, two Managed Elements were defined: ‘pc’ which represents a Node (line 2), and ‘c1’ which represent a Component. The pc node description has a nested Description Property core:hasOS, which is a unary property that limits targeted nodes to those having “ubuntu” as an operating system. The “ubuntu” information is a literal string representing the object value of the core:hasOS Description Property. The c1 component description has a nested Description Property core:onNode, which is a binary property that limits the targeted components to those located on a node having “ubuntu” as an operating system. Here, we have directly set the identifier of the pc node description as an object of the Description Property core:onNode.

6.2.3 Archetype Goals

In Cube, management goals are defined via Archetype Goals, which are represented formally via a dedicated type of Description Properties – we refer to these as Goal Properties (GPs), to avoid confusion. While the informational Description Properties (presented in the previous subsection) are defined as nested XML elements of Managed Elements, Goal Properties are defined via a dedicated XML element called <goals>. The <goals> element is defined as a child element of the <archetype> tag. Within each <goals> tag, the user’s goals are regrouped logically as a set of related goals.

Each Goal Property XML element should be defined using a qualified name, which identifies the actual goal type it represents (exemplified below). The goal type can be selected from an extensible set of predefined goals that Cube can support. A Goal Property element can have the following attributes:

- **s**: the identifier of the Managed Element that represents the goal’s subject.
- **o**: the identifier of the object Managed Element if the goal is of a binary type, or a literal value if the goal is of a unary type.
- **r**: the resolution strategy to be employed. It takes one of the following values: “f”, “fc”, or “c” (representing ‘find’, ‘find or create’, and ‘create’ strategies, respectively, for acquiring managed element instances) as described in section 5.3.2.
- **documentation**: a short description of the goal.

The ‘r’ attribute differentiates Description Properties (DPs) from Goal Properties (GPs): DPs are informational and help identify the Managed Elements to which Goals should apply (subject or object); GPs are management objectives to be attained and hence include a corresponding resolution strategy.

Listing 6.4 shows some examples of Archetype Goals.

Listing 6.4: Example of an Archetype Goals

```
<archetype   id="net.debbabi.cube.demo" version="1.0">
  <goals>
    <goal documentation="components placement">
      <core:onNode s="@a" o="@pc" r="f"/>
      <core:onNode s="@b" o="@tablet" r="f"/>
      <core:onNode s="@c" o="@tablet" r="f"/>
    </goal>
    <goal documentation="self-connecting components">
      <core:connected s="@a" o="@b" r="fc"/>
      <core:connected s="@b" o="@c" r="fc"/>
    </goal>
  </goals>
  <!--element definitions are not shown here -->
</archetype>
```

In this example, we omitted the description of the Managed Elements (identified as "a", "b" and "c" in the listing) and only highlighted goal definitions instead. We regrouped our goals into two sets: the first one is about component placement on nodes, while the second one is about component interconnections. To achieve this, we used two kinds of predefined goals: `core:onNode`, and `core:connected`, respectively. The goal of this Archetype is to ensure that any component that matches the component description identified as 'a' in the archetype is placed on a system platform that matches the node description identified as 'pc'. Similarly, the component descriptions 'b' and 'c' should be placed on platforms that match the 'tablet' node description. Furthermore, we want that any instance that corresponds to the 'a' component description is always connected to another instance that corresponds to the 'b' component description. If no instance is found (following the 'find' resolution strategy), Cube creates a new component instance matching the 'b' component description (following the 'create' strategy). In either case (find or create) Cube connects the acquired instance 'b' to the first instance 'a'. The same objective is also specified between the component descriptions of 'b' and 'c'.

Notice that we have also used `core:onNode` as a Description Property in the previous sub-section. In that case, the purpose was to define the type of components that executed on a certain type of node (e.g. 'pc'). In the example given in this subsection, `core:onNode` is employed for defining an objective that Cube will have to pursue. Namely, the purpose of this goal is to ensure that all instances of component 'a' are placed on nodes of type 'pc'. We notice here that the Goal Property definition (`core:onNode`) utilises the Description Property definition (via a reference to 'pc').

6.2.4 Complete Example

Figure 6.8 shows an example of an Archetype model represented in a graphical language (as used in section 5.3 of chapter 5).

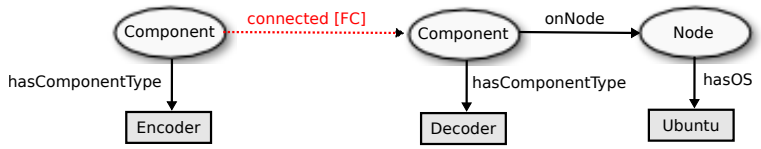


Figure 6.8: Example of an Archetype document in a graph format

This Archetype defines a goal that ensures that any component of type "`Encoder`" is always connected to another component of type "`Decoder`" which is located on a node that has an "`ubuntu`" operating system.

Listing 6.5 shows the equivalent XML document of this Archetype example.

Listing 6.5: Complete Archetype Example in XML format

```
<cube xmlns:core="fr.liglab.adele.cube.core" cube-version="2.0">
  <archetype id="net.debbabi.cube.demo" version="1.0">
    <goals>
      <goal>
        <core:connected s="c1" o="c2" r="fc"/>
      </goal>
    </goals>
    <elements>
      <core:Component id="c1">
        <core:hasComponentType o="Encoder"/>
      </core:Component>
```

```
12        <core:Component id="c2">
13          <core:hasComponentType o="Decoder"/>
14          <core:onNode o="@pc"/>
15        </core:Component>
16        <core:Node id="pc">
17          <core:hasOS o="Ubuntu"/>
18        </core:Node>
19      </elements>
20    </archetype>
21  </cube>
```

The implemented Cube framework prototype only supports Archetypes defined using the XML syntax (and not the graph-based representation). The prototype's XML Archetype parser is detailed in the next sub-section 6.2.5. A graphical editor of the Archetype is not yet implemented, and remains to be achieved in future developments.

6.2.5 Archetype Parser

When the Archetype XML file is loaded by a Cube Autonomic Manager (see next section 6.3), it is parsed to produce its equivalent Java objects (Figure 6.9). In our prototype, we have used the SAX Parser[6], which reads the XML document and returns XML elements progressively. Our Archetype Parser reads the XML elements returned by the SAX Parser and analyses them to identify their nature.

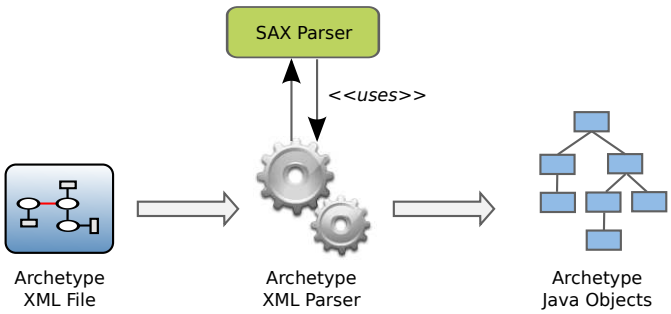


Figure 6.9: Archetype XML Parser

Whenever the parsed XML element does not fit the structure of the Archetype specification an exception of type `ArchetypeParsingException` is fired. Otherwise, the Archetype Parser constructs an Archetype Java Object that corresponds to the parsed XML element. This procedure allows the Archetype Parser to progressively create a Java Object-based representation of the Archetype XML document.

The Archetype Parser is generic as it constructs the same generic Java Objects independently of the nature of the parsed elements (e.g. `Component`, `Node`, for Managed Elements, and `connected` or `onNode` for Properties). The drawback of this solution is the non existence of an offline validation solution that assures the validity of the provided Archetype elements. It is at runtime - when the Autonomic Manager start working - that problems and exceptions can be fired.

Listing 6.6 shows a part of the `Archetype.java` class. It holds references to the different Archetype model elements - including the representation of Managed Elements (`DescriptionElement`

[6]SAX (Simple API for XML) is an event-based sequential access parser API

class – listing 6.7) and their Description Properties (`DescriptionProperty` class) - as well as references to the set of Archetype Goals (`Goal` class). The Archetype has a graph-based data structure. Vertices are Archetype Managed Elements and arcs are Description or Goal Properties.

Listing 6.6: Archetype Java class

```
public class Archetype {

  private String id = "";
  private String description = "";
  private String version = "1.0";
  private String cubeVersion = "2.0";

  private Map<String, DescritpionElement> elements = new HashMap<String,
      DescritpionElement>();
  private Map<String, Goal> goals = new HashMap<String, Goal>();

  public List<DescritpionElement> getDescriptionElements() {
    List<DescritpionElement> result = new ArrayList<DescritpionElement>();
    for (String e : this.elements.keySet()) {
      result.add(this.elements.get(e));
    }
    return result;
  }

  // the remaining code is omitted for simplicity
}
```

The Archetype parser is called by the Cube Runtime Administration Service (see section 6.3.2) when creating and starting a Cube Autonomic Manager. Indeed, each Autonomic Manager should have an Archetype that specifies its self-management mission. As described in Figure 6.9, the Archetype Java Objects are lightweight objects storing the equivalent information provided by the XML file, and conforming to the Archetype model presented in the previous chapter (section 5.3). The beneficial purpose of using Java Objects is to optimize the retrieval of information from the Archetype, as this operation is frequently performed by the Autonomic Manager for ensuring user goals.

Listing 6.7: Java class of the Archetype Description Element

```
public class DescriptionElement {

  private Archetype archetype;

  private String namespace;
  private String name;
  private String id;

  private String description = "";

  private List<DescriptionProperty> descriptions = new ArrayList<DescriptionProperty
      >();

  // the reste of the code is ommitted for simplicity
}
```

The following section provides more details about the runtime part of the Cube project.

6.3 Cube Execution Platform

6.3.1 Overall Architecture

The Cube Runtime Bundle is the most important bundle of the Cube framework. It provides runtime facilities to create and manipulate Cube Autonomic Managers (AMs) that can self-manage software systems. Figure 6.10 shows a simplified UML class diagram of Cube's Runtime platform. We differentiate between core classes - provided by the Cube framework (in grey colour, and detailed just after) - and extension classes - provided by extensions (in white colour, detailed in section 6.4).

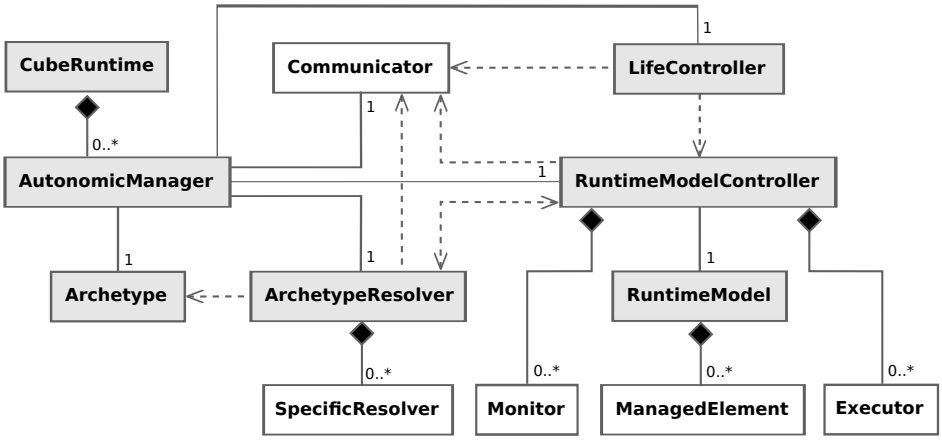


Figure 6.10: UML class diagram of the Cube Runtime Platform

6.3.2 Cube Runtime and its Administration Service

When an OSGi platform starts, the first object created (from the classes depicted in the class diagram in Figure 6.10) is the `CubeRuntime` class. This class is declared as an iPOJO Component and is instantiated automatically at start-up. Its principal role consists in providing an implementation of the Administration Service (detailed in this section), which provides the basic operations for the Cube system in the local OSGi platform.

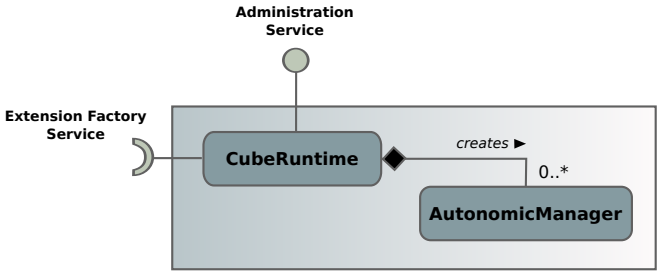


Figure 6.11: Cube Runtime Bundle

Indeed, the CubeRuntime component instance exports one OSGi service called Administration Service and requires one OSGi service called Extension Factory Service (detailed further in section 6.4).

The Administration Service manages (i.e. creates, removes and updates) a set of local Cube Autonomic Managers (Figure 6.11). It also allows other tools to access and retrieve information about the instantiated Autonomic Managers. The required Extension Factory Service allows the Cube Runtime component instance to create the wanted extensions for each Autonomic Manager using the Factory pattern [GHJV95] where a single extension factory is used to instantiate different extension instances for different Autonomic Managers. It also manages the dynamic deployment of new extension factories by handling their arrival and departure.

The following Listing 6.8 depicts a code snapshot from the `CubeRuntime` Java class. We use iPOJO annotations[7] to declare CubeRuntime as an iPOJO Component and to specify which attributes are required and whether this class provides a service or not.

Listing 6.8: Java class of the Cube Runtime implementing the Administration Service

```
@Component
@Provides
@Instantiate
public class CubeRuntime implements AdministrationService {

  @Requires
  private List<ExtensionFactory> extensions;

  private Map<String, AutonomicManager> autonomicManagers;

  private BundleContext bundleContext;

  @Validate
  public void starting() {
    String msg = "\n";
    msg += "\n     _______                  ";
    msg += "\n    /|      |                 ";
    msg += "\n   | | CUBE |... Version:    ";
    msg += "\n   | |______|    " + getVersion();
    msg += "\n   |/______/                  ";
    msg += "\n";
    System.out.println(msg);
  }

  @Invalidate
  public void stopping() {
    System.out.println(" ");
    System.out.println("[INFO] ... Stopping CUBE");

    // Stopping and destroying all the created Autonomic Managers.
    for (String am : this.autonomicManagers.keySet()) {
      stopAutonomicManager(am);
      destroyAutonomicManager(aam);
    }

    System.out.println("[INFO] ... Bye!");
    System.out.println(" ");
  }

  // the rest of the code is omitted for simplicity
}
```

[7] http://felix.apache.org/site/how-to-use-ipojo-annotations.html

The `CubeRuntime` Java class implements the `AdministrationService` interface and is specifies that it provides this service (represented by this Java interface). The iPOJO annotation `@Provides` (line 2) prompts iPOJO to detect automatically the implemented interface and to declare the current Java class as a provider of the corresponding service. The annotation `@Component` (line 1) declares this class as an iPOJO Component, and the annotation `@Instantiate` (line 3) demands iPOJO to instantiate this component when the bundle containing this component is deployed on an OSGi platform and is deemed valid. When the component is started, the starting method (lines 14-23) is called automatically by iPOJO. In this example, this method does nothing special except showing some informative messages on the output console. Similarly, when the bundle containing this component (Cube Runtime Bundle) becomes invalid or stopped, the stopping method of the CubeRuntime class is called automatically by iPOJO. Line 7 declares a list of `ExtensionFactory` objects that are not initialized and that contain no item. This list will automatically receive the set of Extension Factories that are available in the local OSGi platform. This task is transparently ensured by iPOJO because of the `@Requires` annotation specified for this attribute (line 6). Finally, in line 9, we have a map of `AutonomicManager` objects. The key of this map is the URI of each added AM, while the value is a reference to the actual AM object.

6.3.3 Autonomic Manager

The Autonomic Manager's (AM) internal architecture was detailed in section 5.4 of the previous chapter 5. In this section we provide more details about its implementation. In particular, we show how the core modules - Runtime Model Controller, Archetype Resolver, and Life Controller - were implemented. Extension modules - Monitors/Executors, Specific Resolvers, and Communicator - will be discussed in a dedicated section 6.4.

Initialization

AMs are created using the Administration Service as seen in the previous section. Each AM is configured upon creation based on a user-provided configuration file containing mandatory and optional configuration information. The configuration is encoded using the Configuration Java class depicted in Figure 6.12. This class has several core configuration attributes and a set of extension configurations. The different configuration entries are used at creation time. Some entries are used by the Autonomic Manager object itself, others by its associated core constituents (e.g. Archetype Resolver, Life Controller and Communicator), and others by its different declared extensions.

To facilitate the AM instantiation task, we have implemented a tool called *HotDeployer* (section 6.5.2), which takes an XML file representing the Autonomic Manager Configuration and creates dynamically a new AM with the provided settings and extensions.

The following table 6.1 explains the different configuration entries.

The Configuration object has a set of Extension-related configurations (`ExtensionConfig`). It defines the initial list of extensions for the Autonomic Manager. Each extension is identified by its id (namespace) and its version. In addition, each extension has a set of configurations which are provided as a set of key/value entries.

When created, the different constituents of an AM are created and initialized using the provided configuration. If the auto-start configuration is set to true (by default), the AM starts directly upon creation. Otherwise, the user must call the `runAutonomicManager` method of the Administration Service in order to start the AM. When started, the run method of the AM is automatically called by the framework. This method also starts the different extensions associated to the AM. Stopping the AM is achieved via

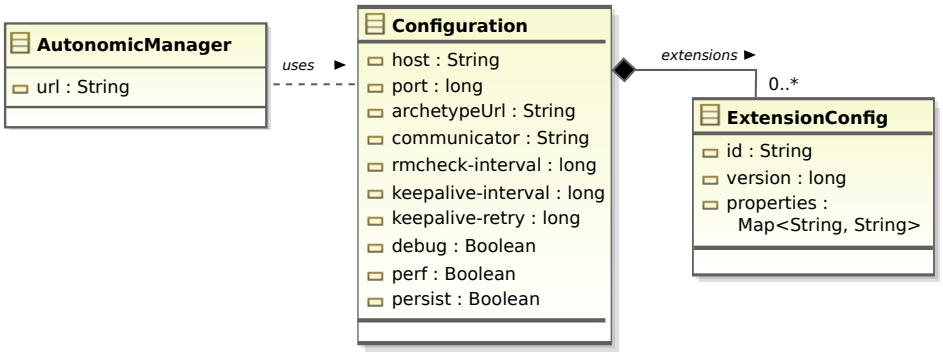


Figure 6.12: The Autonomic Manager's Initial Configuration

the `stopAutonomicManager` of the Administration Service, which will fire the stop method of the AM, and which will stop the different AM extensions.

One of the first tasks that an AM performs is to subscribe as listener to the `Communicator` object after it has obtained a reference to it. This allows the AM to send and receive messages from other AMs. Since the Communicator is an extension and not a core module, this object is provided via an AM extension, namely, the Core Extension (as detailed in section 6.4).

As previously indicated, an AM starts by building the list of its extensions as indicated in its configuration. In particular it loads the Core Extension first, even if it is not explicitly declared in the configuration object. This is because the Core Extension is mandatory for the working of some core functionalities. Figure 6.13 depicts a UML sequence diagram showing the initialization steps of one Autonomic Manager.

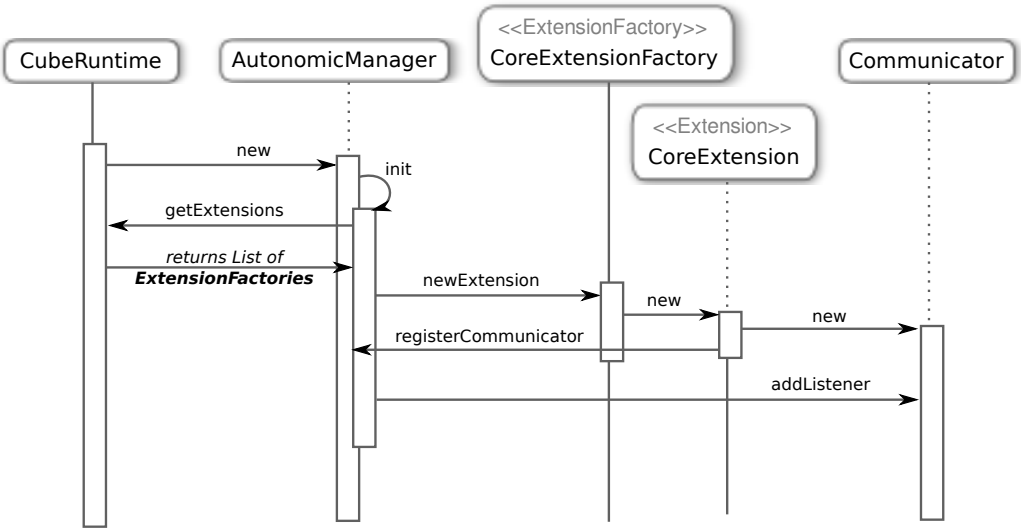


Figure 6.13: Starting Autonomic Manager Sequence Diagram

The Autonomic Manager also creates the different objects of its internal architecture (i.e., Runtime Model Controller, Archetype Resolver and Life Controller). Finally, it parses the Archetype specification to construct the corresponding Java object-based in-memory representation (as detailed before in section 6.2.5).

Table 6.1: Autonomic Manager Configuration Entries

Configuration name	Short Description
host and port	The network host and port used for this Autonomic Manager. They should be available and not used by another Autonomic Manager or by any other external process.
archetypeUrl	The location of the Archetype to be used by the Autonomic Manager. The Archetype can be located locally in the file system, or remotely on another host.
communicator	The name of the communicator implementation to use.
auto-start	Informs the Cube Runtime whether or not the Autonomic Manager should be started as soon as created.
rmcheck-interval	Sets the interval value of the checking cycle of the Runtime Model. During each cycle, Cube's Archetype Resolver checks for INVALID instances in the Runtime Model and tries to resolve them.
keepalive-interval	Related to the functioning of the Life Controller module of the Autonomic Manager. It sets the interval for checking the liveliness of neighbour Autonomic Managers.
keepalive-retry	Sets the number of retries that the Life Controller should perform before considering a remote Autonomic Manager as failed.
debug	Activates or deactivates debug messages in the console. When set to true, all the details about the working of the Autonomic Manager are shown in the console.
perf	If set to true, Cube generates an evaluation summary file providing data on the performance of the Autonomic Manager.
persist	If set to true, the Runtime Model instances are persisted to the hard disk drive. When the Autonomic Manager stops and is restarted again, it loads the previous persisted Runtime Model instances. However, this feature is not implemented in the current prototype.

6.3.4 Runtime Model Controller

The Runtime Model Controller (RMC) is a container of the local Managed Element instances (which make-up the AM's local Runtime Model – section 5.4.2). It also provides a distributed, transparent API that allows retrieving information about remote instances located on neighbouring Autonomic Managers. As shown in Figure 6.10, the RMC is used by several internal objects of the Autonomic Manager. It exposes several methods allowing the manipulation of the local Runtime Model. Listing 6.9 shows an example of one such method - `getProperty(...)`, which can be used to retrieve the property value of a Managed Element instance, identified by its universal unique identifier UUID. If the concerned ME instance is managed by the current AM, the method returns directly the value of the wanted property (lines 4 to 6); or null if the property does not exist. In the case the instance is managed by a remote AM, the method sends a message to the remote AM in order to retrieve the property value (lines 8 to 25).

Listing 6.9: Getting property value of a Managed Element using the Runtime Model Controller class

```
1 public class RuntimeModelControllerImpl implements RuntimeModelController {
2   // ...
3   public String getProperty(String managed_element_uuid, String name) {
```

```
4       ManagedElement instance = getLocalElement(managed_element_uuid);
5       if (instance != null) {
6         return instance.getProperty(name);
7       } else {
8         String am = autonomicManager.getExternalAMUri(managed_element_uuid);
9         if (am != null) {
10        CMessage msg = new CMessage();
11        msg.setTo(am);
12        msg.setObject("runtimemodel");
13        msg.setBody("getProperty");
14        msg.addHeader("uuid", managed_element_uuid);
15        msg.addHeader("name", name);
16        try {
17          CMessage resultmsg = sendAndWait(msg);
18          if (resultmsg != null) {
19            if (resultmsg.getBody() != null) {
20              return resultmsg.getBody().toString();
21            }
22          }
23        } catch (TimeOutException e) {
24          e.printStackTrace();
25        }
26        } else {
27          System.out.println("No such managed instance: " + managed_element_uuid);
28        }
29      }
30      return null;
31    }
32    // ...
33  }
```

Technically speaking, the exemplified method takes two parameters. The first one is the universal identifier (UUID) of the targeted Managed Element instance, while the second one is the name of the desired property. In line 4, we start by retrieving the instance object using the `getLocalElement(...)` method part of the same class. This method looks inside the local Runtime Model for an instance with the given UUID. If found (line 5), it directly calls the `getProperty(...)` method of the found ME instance and returns its value (or null if no property with the given name is found). When no instance of the given UUID is found, the method tries to check if any existing UUID is saved by the Autonomic Manager as a remote instance. If so, by calling the `getExternalAMURi(...)` method it can retrieve the URI of the remote AM in which the targeted ME instance is located (line 8). We use the remote AM's URI to communicate with the referenced AM and hence to retrieve the desired property value of the instance managed by this AM. Therefore, the URI is used as destination information of the message to be sent using the `setTo(...)` method of the `CMessage` object (line 11). When the remote AM receives this type of messages, it checks the "Object" of that message to determine which internal module the message is destined for. In our case, we want the message to be transmitted to the Runtime Model Controller - this is why we indicate "*runtimemodel*" as the targeted object (line 12)., When receiving the message, the remote RMC checks the body of the message to determine which information is requested. In the case of this method, it is the property value that we want to get (line 13). To effectively respond to this request, the remote RMC should also receive more information about the property, in particular its UUID and name. This information is passed as message headers (lines 14-15). Finally, the original RMC calls the `sendAndWait(...)` method to send the message to the remote AM. This method is blocking, meaning that the RMC will wait until the remote RMC responds to its request. Otherwise,

a `TimeOutException` is fired. In this method, we automatically set the "ReplyTo" header so that the remote AM and its RMC respond to the requesting RMC. Listing 6.10 shows the way in which the `RuntimeModelController` class handles received messages.

Listing 6.10: Handling remote requests within the Runtime Model Controller

```
1  public class RuntimeModelControllerImpl implements RuntimeModelController {
2    // ...
3    protected void handleMessage(CMessage msg) throws Exception {
4      if (msg != null) {
5        if (msg.getBody() != null) {
6          if (msg.getBody().toString().equalsIgnoreCase("getProperty")) {
7            Object uuid = msg.getHeader("uuid");
8            Object name = msg.getHeader("name");
9            String p = null;
10           if (uuid != null && name != null) {
11             p = getProperty(uuid.toString(), name.toString());
12           }
13           CMessage resmsg = new CMessage();
14           resmsg.setTo(msg.getFrom());
15           resmsg.setCorrelation(msg.getCorrelation());
16           resmsg.setObject(msg.getObject());
17           resmsg.setBody(p);
18           try {
19             getCubeAgent().getCommunicator().sendMessage(resmsg);
20           } catch (CommunicationException e) {
21             e.printStackTrace();
22           } catch (IOException e) {
23             e.printStackTrace();
24           }
25         } // else
26         //   if (msg.getBody().toString().equalsIgnoreCase("addProperty"))
27         //   {...}
28       }
29     }
30   }
31   // ...
32 }
```

The `handleMessage(...)` method handles messages received from remote AMs. All messages arriving to this function have "*runtimemodel*" as object. At this step, we filter the messages depending on their body, which contains the requested function. The listing above shows only the case when "*getProperty*" is sent from the `getProperty(...)` function of the remote RMC as shown in listing 6.10. In lines 7 and 8 we retrieve the transmitted UUID and property name and use them to get the value of the property identified by the name from the local ME instance identified by its UUID (line 11). The found value (or null if not found) is returned to the calling remote AM. We set the correlation header of the respond message to be the same as the request message (line 15) so that the original caller waiting for a response message can identify it among other received messages. We use the `sendMessage(...)` method of the Autonomic Manager's Communicator to send the response message, without waiting for a response. More information on the Cube's messaging system can be found in section 6.4.2 when talking about the Communicator extension point.

6.3.5 Data Structure and Implementation for the Archetype Resolver

The Archetype Resolver (AR) is the module responsible for automatically resolving problems when they appear in the Runtime Model (and hence in the managed system). It is tightly linked with the Runtime Model Controller and has a reference to the parsed Archetype model containing the user-specified goals.

Technically speaking, the Archetype Resolver has two working modes: active and passive. In the active mode, the Archetype Resolver is implemented as a Java Thread which periodically inspects the Runtime Model for invalid Managed Element (ME) instances. If an invalid ME instance is found, the AR launches the resolution process to (re)validate the instance. In the passive mode, the AR is subscribed as a listener of Runtime Model changes. Hence, any changes in the local Runtime Model elements are notified to the AR, which then launches the resolution process if any of the changes involve ME instances becoming invalid. In our prototype implementation, we use a mix of the two modes – active and passive.

Data Structure and Architecture

Figure 6.14 depicts the overall design of the Archetype Resolver. It highlights its dependencies with the rest of the AM's modules, and shows its internal data structure. The `ArchetypeResolver` class has references to the `RuntimeModelController` and `Archetype` objects via the `AutonomicManager` object, and it stores a set of `SpecificResolver` objects provided by extensions (see section 6.4). At runtime, the `ArchetypeResolver` object creates and removes `ResolutionGraph` objects as needed.

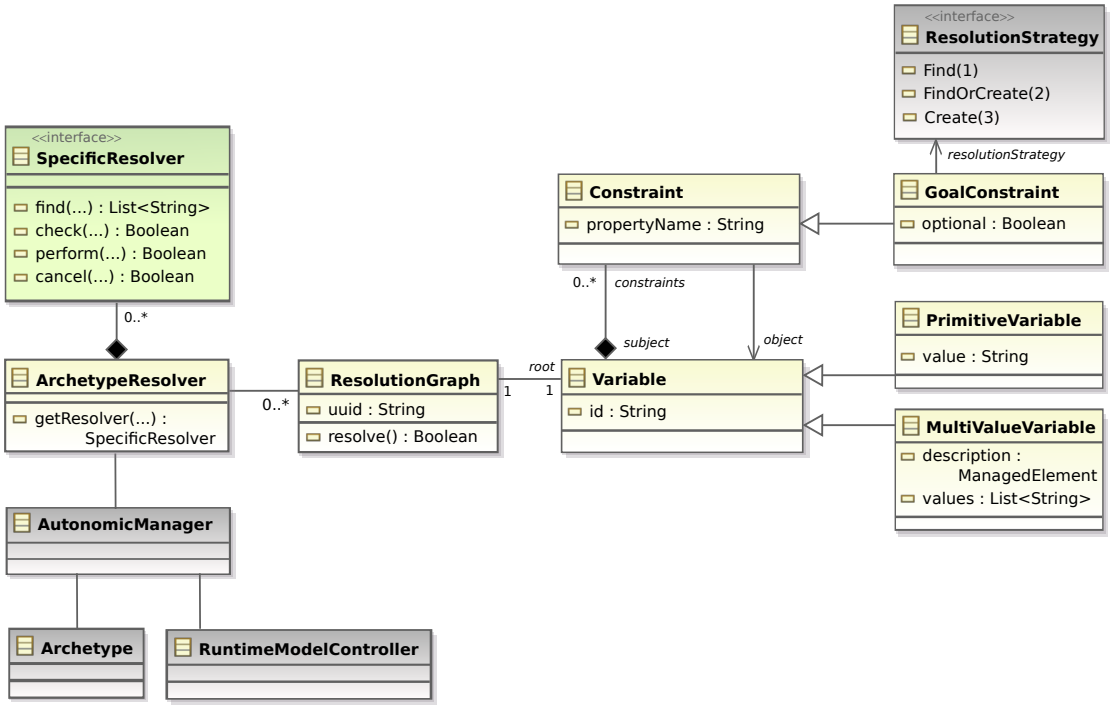


Figure 6.14: Archetype Resolver Design

Indeed, a `ResolutionGraph` object is created for each invalid Managed Element instance that must be resolved. The `ResolutionGraph` object is comprised of a set of `Variable` and `Constraint`

objects – variables represent graph nodes and constraints graph edges. Variables may have only one value corresponding to a Primitive Element from the Archetype, or multi-values corresponding to the different checked Runtime Model instances that conform the description wanted by this variable (correspond to Managed Element form the Archetype). The graph's Variable and Constraint objects are not directly stored in the `ResolutionGraph` object itself. Instead, the `ResolutionGraph` object only contains a reference to the graph node (Variable) representing the invalid Managed Element, which must be resolved; this is referred to as the root variable and represents the graph's root node. Each `Variable` object can be associated to a set of `Constraint` objects (graph arcs). Each constraint has a reference to another `Variable` object. Following this path, the resolution graph is constructed and can be easily explored. As we have discussed in the previous chapter, we have adopted a Constraints Satisfaction Problem (CSP) oriented approach to enable Cube to achieve user goals. The problem is specified as a constraints graph (also referred to as Resolution Graph). This graph contains a root node (Variable object) – representing the Managed Element that must be validated – and all the other nodes (`Variable` objects) representing Managed Elements that are connected via Constraints (binary or unary) to the root node, directly (via Goal Constraints corresponding to Archetype Goal Properties) or indirectly (via Constraint objects corresponding to Archetype Description Properties). The corresponding data structure is comprised of the three aforementioned Java classes: `ResolutionGraph`, `Variable`, and `Constraint`.

Each `ResolutionGraph` object has a universal unique identifier (UUID). This is important because the overall `ResolutionGraph` can be comprised of multiple sub-paths distributed over several Autonomic Managers. In other words, when constructing a local Resolution Graph to find a viable solution, the resolution process can span beyond the local solution space, having to look for remote instances in other Autonomic Managers. Hence the graph is split between two or more Autonomic Managers, which must work together to find an overall solution. Here, UUIDs are used to identify clearly the original `ResolutionGraph` and its sub-graphs.

As mentioned before, a `Variable` object represents one graph node. It can provide a description of a Managed Element – either the Managed Element to validate - the graph root - or other Managed Elements related to the root ME via binary constraints. In this case, an instance of `MultiValueVariable` is used. The different possible values correspond to UUID identifiers of the checked Runtime Model instances. A Variable can also represent a Primitive Element, in which case it simply provides a literal value. In this case, an instance of `PrimitiveVariable` is used.

A Variable has a set of related Constraints. The corresponding Constraint objects are the equivalent of Archetype Properties (Description Properties and Goals). Indeed, when constructing the Resolution Graph, the Archetype Resolver analyses the current invalid instance and the user-provided Archetype model to retrieve the user-declared goals for this instance.

In the following, we detail the manner in which the resolver algorithm works, relying on the presented data structure. While we differentiate between local resolution and distributed resolution, these two cases are implemented in the same Java class and applied the same way for either local or remote resolution. It depends on runtime context and the user-provided Archetype.

Local resolution

Let's consider the Archetype example depicted in Figure 6.8, which imposes that any component instance of type "Encoder" should always be connected to a component instance of type "Decoder" that is located on a node that has "Ubuntu" operating system.

We suppose that a component instance of type "Encoder" is created by an external entity, discovered by a Cube monitor and added to the local Runtime Model. As soon as it is being notified by this change, the *Archetype Resolver* (AR) builds a Resolution Graph to validate this newly created component instance.

It first creates a `MultiValueVariable` object representing the new component and makes it the root node of the Resolution Graph (e.g., 'v1' node in Figure 6.15). From the Archetype, the AR finds that the description of this component instance fits the subject description of the "connected" goal. For this reason, it adds to the graph a `GoalConstraint` that has "connected" as `propertyName`, and associates it to the root variable containing the UUID of the ME instance to validate (the UUID of an instance is represented by `#instance` shortcut in the following Figures). Figure 6.15 represents different steps in the resolution of this new instance. In (t1), the AR builds a complete Resolution Graph from the Archetype. Indeed, from this graph, it results that the AR should guarantee that the component instance `#Comp2` (v1 node) should be connected to another component instance (not yet identified, corresponding to the v2 node), which must conform to the associated description.

The resolution algorithm tries to acquire an adequate instance for the v2 variable so that it satisfies the two constraints - has Component Type "Decoder"; and, located on Node which has "Ubuntu" operating system. The acquisition procedure will depend on the resolution strategy specified via the "connected" constraint – 'find' and/or 'create'.

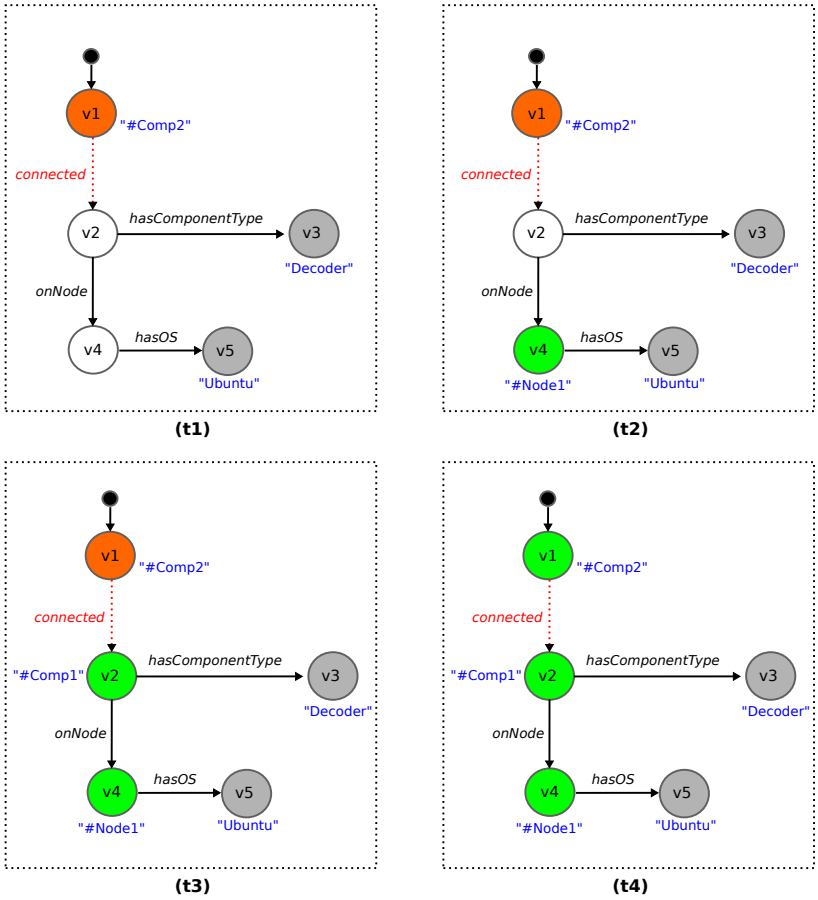


Figure 6.15: Example of Local Archetype Resolver Resolution Graph. Red circle represents invalid component instance. Green circle represents valid instance. Grey circle represents primitive variable. Red arrow represents a goal. And black arrow represents Description Property.

For each goal constraint (in the example there is only one - "connected"), the resolver tries to find an acceptable value for the related variable using a *backtracking* algorithm. When a candidate value does not

satisfy a variable's constraints, it is tagged as non-viable and stored in the variable's history.

The Archetype Resolver calls different methods of Specific Resolvers in order to find candidate variable values, to check the found values, and to perform the resolved goal (see Specific Resolver Extension Points in section 6.4.2).

More precisely, finding values for a variable is done by calling the find method of the description constraint for that variable. This is only possible if the object (second) variable of the constraint has already a value. In Figure 6.15, to find a candidate value for v2, the resolver should first find a value for the v4 variable. In (t2), v4 has a candidate value - a node instance `#Node1` with "Ubuntu" operating system. Now, to find a value for v2, the resolver calls the find method of the "onNode" Constraint, where the object variable already has a value (`#Node1`). The find method is implemented by a Specific Resolver provided by the core extension (see Figure 6.14). This Specific Resolver looks for Component instances that are located on `#Node1` and that have a matching description (has Component Type "Decoder"). In (t3), the resolver has found matching component instance `#Comp1`. Hence, it now checks whether applying the "connected" goal constraint (by calling the "perform" method) will not break any constraints of v2. If this is the case, the performed constraint is validated and `#Comp2` and `#Comp1` are subsequently connected (t4).

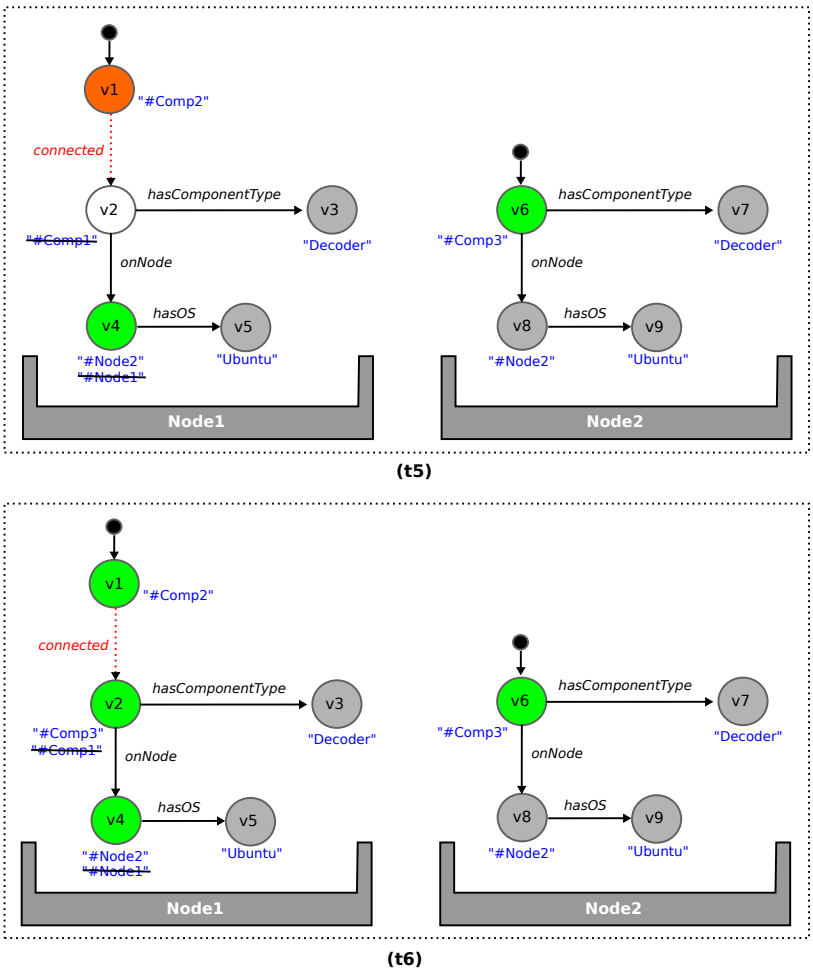


Figure 6.16: Example of distributed Archetype Resolver Resolution Graph

Distributed Resolution

Now, let's imagine that the found component instance `#Comp1` does not satisfy v2's constraints. In this case, it is marked as a non-viable solution and stored in v2's history stack. The backtracking algorithm tries to find another component instance located on `#Node1`. However, no such component is found (Figure 6.16–(t5)). In this case, the `#Node1` value of v4 is also marked as non-viable, and a new candidate value for v4 is needed. The resolver finds a new value corresponding to a remote node - `#Node2`. For simplicity, in this example we do not show the constraint "inScope" related to v4, which enables the resolver to find node instances that belong to a particular scope. At this level, as the `#Node2` is managed by a remote AM, the Resolution Graph must grow and spread onto this AM's platform in order to find a valid value for the v2 variable. Note that on the remote AM that manages `#Node2` a new graph is formed with a variable v6 at its root. Here, v6 is the equivalent of v2 in the initial AM located on `#Node1`. When a value is found in Node2 for the v6 variable the resolution algorithm of the local AM first checks this value. If all v6's constraints are satisfied the "connected" constraint is performed between the two remote instances (t6) – note that v2 becomes valid (green in the figure) and has the same UUID as v6 (`#Comp3`).

6.4 Extensions

6.4.1 Overview

The Cube framework is designed considering high extensibility and reusability requirements. The internal architecture of Cube's Autonomic Manager (presented in section 5.4.1) promotes such features by decoupling its constituent modules explicitly and providing each one of them with very specific and well-defined roles. This allows for user-provided implementations of these modules to be integrated into the Autonomic Manager in order to extend its capabilities and to allow taking into account user-specific goals.

Such customized modules represent extension points of the internal architecture of the Autonomic Manager. When a developer intends to implement new extensions, (s)he must develop one or more of these extension points and register them with the related Autonomic Manager (via the AM's configuration). Extension points are discussed in the next sub-section 6.4.2.

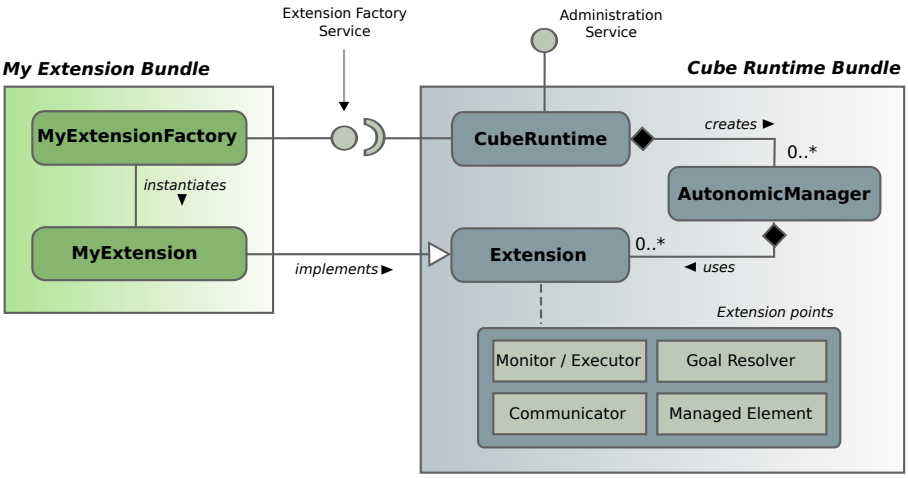


Figure 6.17: Extension mechanism

Figure 6.17 shows the integration of user-provided extensions with the Cube Runtime. User provided extensions are packaged into OSGi bundles. This type of bundle should contain at least two classes: the first one implements the `ExtensionFactory` Service interface, provided by the Cube Runtime Bundle; and, the second one implements the Extension class (or extends the `AbstractExtension` class), also provided by the Cube Runtime Bundle.

The `CubeRuntime` component defines a service reference (requirement) to all components that implement the `ExtensionFactory` service. Hence, at any time, all the references to the available `ExtensionFactory` implementations are saved in a list within the `CubeRuntime` component. The `CubeRuntime` component also implements (provides) the `ExtensionFactory` service. Hence, when the user creates a new Autonomic Manager (via the Administration Service) the `CubeRuntime` component reads the AM's provided configuration, and for each declared extension, it calls its corresponding `ExtensionFactory` component to create a new instance of that extension and add it to the new Autonomic Manager. Indeed, the Extension bundle implements a factory-based pattern to instantiate extension objects used for each Autonomic Manager.

6.4.2 Extension Points

As depicted in Figure 6.17, extension points are of four types. The following table 6.2 provides a short description of each one of the possible Autonomic Manager extension points.

Table 6.2: Extension Points of Cube Autonomic Manager

Extension Point name	Short Description
Managed Element	This extension point allows the introduction of new types of Managed Elements to the Cube system, representing modelled elements for new types of managed resources. Following the Cube meta-model presented in chapter 5, such new modelled elements are part of the M2 layer of Cube modelling layers (see previous chapter, section 5.2).
Monitor/Executor	This extension point allows the introduction of new monitors and executors specific to a technology of the managed system. Before defining such monitor/executor extensions we have to have already defined Managed Element models corresponding to the system's managed resources – by either defining new Managed Element extensions or by reusing available models that are suitable for representing the managed resources. Monitors/Executor will ensure the mapping between the managed element in the real system and the modelled element in Cube's Runtime Model.
Specific Resolver	This extension point allows the introduction of specific resolvers capable of handling new types of goals.
Communicator	This extension point allows the introduction of new implementations for the Communicator module.

For each of this extension points, we provide Java interfaces or abstract classes that users should implement or extend. We also provide a simplified API that allows users to register their extension points to with the targeted Autonomic Manager. We detail each type of extension point over the following subsections.

Managed Element Extension Points

Users should extend the `ManagedElement` class to introduce new modelled elements into the system. This means implementing a new class that extends `ManagedElement` while respecting some restriction and guidelines:

- *No Java attributes*. When the user wants to add functional or non-functional properties and attributes to the new modelled element, they should do so by defining setter (set) and getter (get) qualifiers. When implementing a set method, the user should call the `addProperty(...)` method of the super class `ManagedElement`, which adds the attribute to the list of the `ManagedElement`'s properties. For all the get methods, the user can retrieve the value of a given property by calling the `getProperty(...)` method of the super class. This guarantees that all the new modelled elements will share the same data structure (provided by the `ManagedElement` super class).
- *No Java references*. If there is a need to reference other modelled elements, users should avoid using direct Java references (pointers). In the Cube meta-model, references between modelled elements are defined using UUID identifiers. Hence, similarly to the properties, we should call the special-purpose methods of the `ManagerElement` super class in order to add references or to get already added references..

Listing 6.11 shows the ManagedElement class provided by the Cube framework.

Listing 6.11: Managed Element Java class

```java
public interface ManagedElement {

  public static final int UNMANAGED = -1;
  public static final int INVALID = 0;
  public static final int VALID= 1;

  public String getName();
  public String getNamespace();

  public String getUUID();
  public String getAutonomicManager();
  public String getUri();
  public int getState();

  public List<Property> getProperties();
  public boolean hasProperty(String name);
  public String getProperty(String name) ;
  public boolean addProperty(String name, String value) throws PropertyExistException
      , InvalidNameException;
  public String updateProperty(String name, String newValue) throws
      PropertyNotExistException;
  public boolean removeEmptyProperties();

  public List<Reference> getReferences();
  public Reference getReference(String name) ;
  public boolean hasReference(String name);
  public Reference addReference(String name, boolean onlyOne) throws
      InvalidNameException;
  public boolean removeEmptyReferences();
  public boolean removeReferencedElement(String ref);
}
```

Lines 3 to 5 define static attributes representing the different states that a Managed Element instance can take. We can retrieve a Managed Element instance's state by calling the `getState(...)` method (line 13).

All the methods of this interface (except for `getName(...)` and `getNamespace(...)`) are implemented within the `AbstractManagedElement` class (Figure 6.18), which is part of the Cube framework. This abstract class can be used by developers to avoid having to re-implement all the interface methods, and to guarantee the aforementioned restrictions when introducing new modelled elements. For each new modelled element class, the `getName(...)` method (line 7) should return the name of the new modelled element. Similarly, the `getNamespace(...)` method (line 8) should return the namespace associated with the extension to which the modelled element belongs.

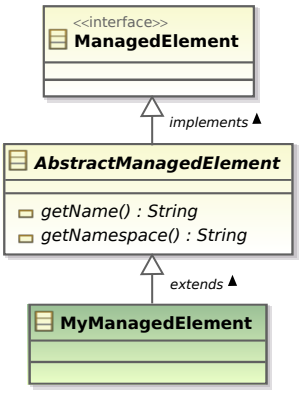


Figure 6.18: Abstract Managed Element class

The methods defined between lines 10 and 13 return basic runtime information about the Managed Element instance, like its universal identifier (line 10), the URL of its Autonomic Manager (line 11), the URL of the current instance (line 12), and the instance's state (line 13). The URL of a Managed Element instance follows the following pattern: *<Autonomic Manager URL> + </Name of the Managed Element type> + </UUID of the Managed Element instance>* (e.g., cube://localhost:38/component/6ece1fd8-301f-4b4a-9f7b-65e56174fb75).

The methods defined between lines 15 and 20 are dedicated to property handling. They allow adding, retrieving, updating, and removing Managed Element properties. Finally, the methods defined between lines 22 and 27 are dedicated to reference handling. All these methods are implemented in the AbstractManagedElement class, provided by the Cube framework.

Monitor/Executor Extension Points

Figure 6.19 shows how user-specific Monitors and Executors are integrated into Cube's Autonomic Manager architecture.

The central piece of this architecture is the `RuntimeModel` which contains the different Managed Element instances. The `RuntimeModel` is handled by a `RuntimeModelController` object, which ensures the model's integrity, manages the different states of modelled element instances, and provides a change notification mechanism. New Monitors and Executors are attached to the Runtime Model Controller.

The `Monitor` interface, and its direct implementation class `AbstractMonitor`, represent the monitoring extension point where developers can integrate new technology-specific monitors. The role of

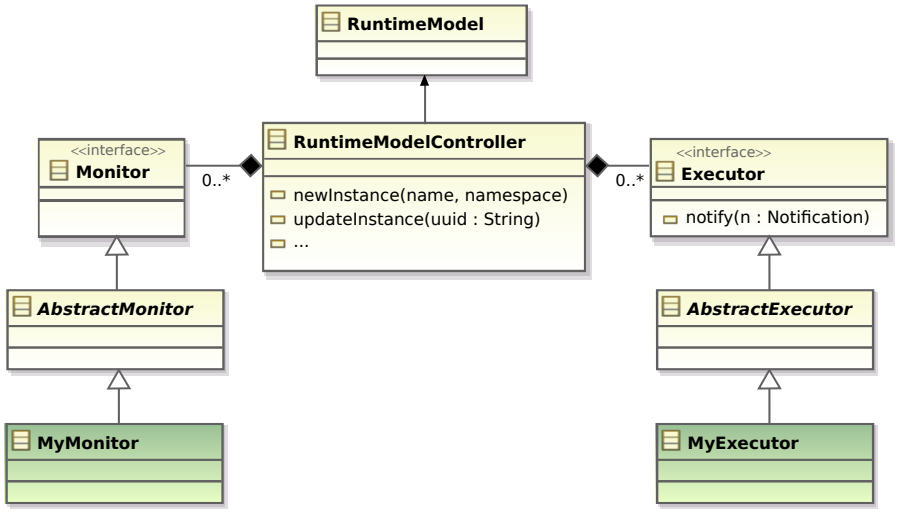


Figure 6.19: Adding new Monitors / Executors

a Monitor component instance is to update the local Runtime Model dynamically, according to changes that occur in the software system part that the corresponding AM is managing.

Similarly, the Executor interface and its implementation class AbstractExecutor allow to observe the Runtime Model and to apply the necessary changes into the real managed software system. Executors are subscribed as Runtime Model listeners to the Runtime Model Controller object, which manages model-related notifications. Hence, whenever there are changes in the Runtime Model, the Runtime Model Controller notifies executors about this update.

Different levels of notifications are supported. Executors can be customized to receive only the notification types they are interested in. These can range from all notifications (about any change in the Runtime Model), to specific notifications about the value changes of one particular property of a particular Managed Element instance.

To create new modelled element instances in the Runtime Model, Monitors call the newInstance(...) method of the Runtime Model Controller object. This method takes two parameters: the name and the namespace of the Managed Element to create. Initially, the created instance is unmanaged by the Autonomic Manager, which means that changes occurring in this new instance will not fire notifications and not trigger the resolution process. This setting is important for allowing the Monitor to prepare the newly created instance before validating it. In particular, the Monitor can update the instance's properties and references so as to fit the monitored system's state.

Specific Resolver Extension Points

Specific Resolvers are introduced to allow the Generic Archetype Resolver find viable solutions that meet specific archetype constraints. When the Archetype Resolver launches the resolution process it must perform various resolution tasks, like looking for a value for a visited variable or performing the related property. For each such task, the Archetype Resolver algorithm looks for the adequate Specific Resolver and calls its adequate method for performing the task.

To develop a new Specific Resolver, users should implement the `SpecificResolver` interface (or, ideally, extend the `AbstractSpecificResolver` class).

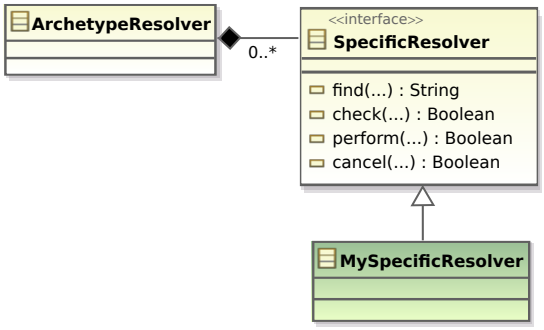


Figure 6.20: Adding a new Specific Resolver

Specific Resolvers make intensive use of the Runtime Model Controller API (local and distributed) in order to retrieve information about Managed Element instances in the Runtime Model. The ME instance UUIDs are then used as values for the Resolution Graph Variables.

The SpecificResolver interface (listing 6.12) show the detail of each method that users should implement to integrate their specific resolvers to the Cube system. In the following we give a short description of each method.

Listing 6.12: Registering a Specific Resolver

```
public interface SpecificResolver extends ExtensionPoint {

  List<String> find(ManagedElement element, ManagedElement description);

  boolean check(ManagedElement element, String value);

  boolean perform(ManagedElement element, String value);

  boolean cancel(ManagedElement element, String value);
}
```

- *find(...)*. This method takes two parameters "element" and "description" of type `ManagedElement`. The first parameter corresponds to the *Runtime Model instance* in which we look for possible solution values; while the latter parameter corresponds to the description of the element which we are looking for. The Archetype Resolver ensures that the "element" instance is a local instance; otherwise it delegates the finding procedure to the Autonomic Manager hosting the instance (as described in 6.3.5). Developers implementing new specific resolvers should query the Runtime Model Controller to get information about the "element" instance and return solution values if found (returns a list of their UUID identifiers).
- *check(...)*. This method is called by the Archetype Resolver to check the *Runtime Model instance* given as parameter ("element") which has a property or a reference given in the "value" parameter.
- *perform(...)*. This methods performs the constraint in the Runtime Model instance given as parameter ("element"). It adds the property or the reference given in the "value" parameter to the target instance.
- *cancel(...)*. This methods do the opposite of the perform method. We remove the property or the reference given in the "value" parameter from the list of properties or references of the target Runtime Model instance given in the "element" parameter.

Appendix B.3 gives an implementation detail of the "connected" Specific Resolver.

Communicator Extension Points

The *Communicator* is the Autonomic Manager module that is responsible for exchanging messages between different Autonomic Managers within the Cube system. As discussed in the previous chapter, section 5.4.6, all Cube Autonomic Managers use the same communication protocol. The internal modules of the Autonomic Manager do not have to worry about the manner in which this protocol is implemented. We provide the same communication interface that encapsulates any implementation differences. This is why we can have different communication patterns for the Cube System. Using one implementation pattern or another will depend on the degree of distribution, physical network and constraints of the managed system. For instance, peer-to-peer patterns will fit mostly in large scale, mobile and/or ad-hoc network scenarios; while point-to-point TCP/IP messages or publish/subscribe patterns may be best suited to well-connected and more reliable networks, like local area networks (LANs) or the Internet.

Technically speaking, in our Cube prototype we have proposed direct socket-based communication over the TCP/IP protocol. This could be easily replaced with other approaches, by providing different Communicator implementations. Any such user-specific implementation should implement the Communicator interface provided by the Cube framework (Figure 6.21). This interface defines two simple methods:

- `send(CMessage msg)`: send the message provided as parameter to a destination set in the message itself (see section 5.4.6 in the previous chapter);
- `addListerner(MessagesListener listener)`: add a listener for received messages.

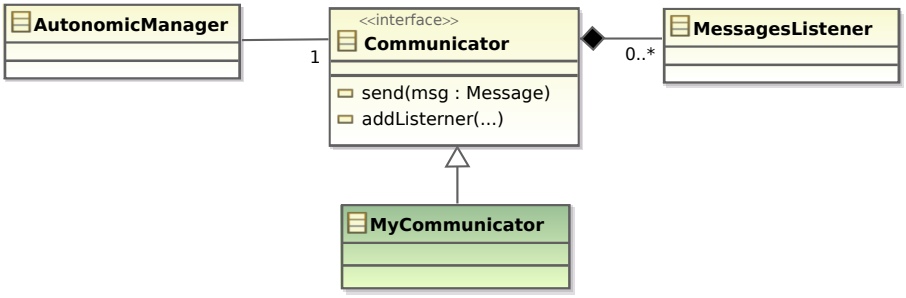


Figure 6.21: Adding new Communicator

When sending a message, the user should fill-in the different message headers so that it will be correctly delivered to its destination.

6.4.3 Deploying and using extensions

All the implemented extension points that adhere to a common technology or target a particular use should be bundled into one OSGi bundle; they will then be identified by the same namespace. The Java interfaces and classes related to the extension itself, and not to the extension points, are as follows (note that the names given here are simply examples):

- `MyExtensionFactory` extends Cube framework's `ExtensionFactory` interface. It does not add any new method, only static final attributes containing the extension's name and namespace.

- `MyExtensionFactoryImpl` implements `MyExtentionFactory` interface. It is used as a Factory to instantiate MyExtension instances, which are used by Autonomic Managers. This class should be declared as an iPOJO Component (using the `@Component` annotation), which provides the implemented service (using the `@Provides` annotation) and which is instantiated automatically when the bundle starts (using the `@Instantiate` annotation).
- `MyExtensionImpl` extends `AbstractExtension` and contains Java code for initializing the extension and registering the different extension points.

In addition, an extension's bundle also contains implementations of the different extension points that were explained before. It is recommended to regroup classes that belong to the same extension point within the same package (e.g. model for classes implementing the Managed Element Extension Point, monitorsExecutors for classes implementing the Monitors/Executors Extension Point, resolvers for classes implementing the Specific Resolvers Extension Point, and communicator for classes implementing the Communicator Extension Point).

Finally, Extensions can be added or removed dynamically from any Autonomic Manager using the Administration Service, or via user-friendly tools like the HotDeployer (section 6.5.2).

6.4.4 Extensions provided by the Cube Framework

Core Extension

The Core Extension is the most important extension provided with the Cube framework, since it implements and provides the core extension points that this thesis contributed to the data mediation domain; it can also be applied more generally to software systems constituted of distributed stateless components. The Specific Resolvers implemented and the proposed concepts like Scope and Master provide a basis for the distributed searching mechanism of the Archetype Resolver. These modelled elements and their associated Specific Resolvers can be re-used for other domains where the managed system is distributed over several execution nodes.

Table 6.3 provides a summary of the Managed Elements included in the Core Extension, together with their properties and references (as shown in Figure 5.6 of the previous chapter).

Table 6.3: Manaed Elements contributed by the core Extension

Managed Element	Property (Type)	Reference (Type)
Component	componentId (String)	node (Node)
	componentType (String)	inputComponents (Component)
		outputComponents (Component)
Node	nodeId (String)	scope (Scope)
	nodeType (String)	components (Component)
Scope	scopeId (String)	master (Master)
		nodes (Node)
Master		scopeLeaders (Scope)

Each of these modelled elements was detailed in section 5.2.3 of the previous chapter.

We have also implemented a set of Specific Resolvers to ensure goals defined for the aforementioned Managed Elements. Table 6.4 provides a summary of these Specific Resolvers, which can be applied to any Managed Element.

Table 6.4: ManagedElement-related Specific Resolvers

Resolver	Subject Type	Object Type	Short Description
inAutonomicManager	ManagedElement	String	The object is the URL of the Autonomic Manager in which the subject Managed Element is or should be located.
hasProperty	ManagedElement	String	Checks if the subject Managed Element instance has the property given as object.

Table 6.5 provides a summary of Specific Resolvers applied to the Component modelled element.

Table 6.5: Component-related Specific Resolvers

Resolver	Subject Type	Object Type	Short Description
hasComponentType	Component	String	Checks if the Component given as subject has the type given as object.
hasMaxInputComponents	Component	Integer	Checks if the Component given as subject has at most the number of input components given as object.
connected	Component	Component	Checks if the two components given as subject and object are connected. When used as a goal, the two components will be connected.
hasSourceComponent	Component	Component	Checks if the Component given as subject has as input component the Component given as object. When used as a goal, the two components will be connected.
onNode	Component	Node	Checks if the Component given as subject is on node given as object. When used as goal, the component will be placed on the given node.

Table 6.6 provides a summary of Specific Resolvers applied to the Node modelled element.

Table 6.6: Node related specific resolvers

Resolver	Subject Type	Object Type	Short Description
hasNodeType	Node	String	Checks if the Node given as subject has the type given as object.
hasComponent	Node	Component	Checks if the Node given as subject has a Component of the object type in its components list.
inScope	Node	Scope	Checks if the Node given as subject is a part of the Scope given as object. When used as a goal, the Node will be included into the Scope.

Table 6.7 provides a summary of Specific Resolvers applied to the Scope modelled element.

Table 6.7: Scope related specific resolvers

Resolver	Subject Type	Object Type	Short Description
hasScopeId	Scope	String	Checks if the Scope given as subject has the identifier given as object.
hasNode	Scope	Node	Checks if the Scope given as subject has the Node given as object in its nodes list.
controlledBy	Scope	Master	Checks if the Scope given as subject has the master given as object.

In this Core Extension, we have also provided the default implementation of the Communicator interface. This implementation is based on network Sockets and uses implements a Client/Server communication pattern to exchange messages. Each Autonomic Manager is considered to be a server that accepts client connections from other Autonomic Managers; it is also considered to be a client when it connects to other Autonomic Managers for requesting some information.

No specific Monitors/Executors were implemented in the Core Extension for these core modelled elements. This is left to be provided by technology-specific extensions, as we have done for the Cilia mediation framework (next sub-section).

The Core Extension is identified by the "`fr.liglab.adele.cube.core`" namespace.

Cilia Extension

Cilia is a data mediation framework [GMD+11] that we have employed for validating our Cube framework prototype (see next chapter). We have implemented a Cube extension for Cilia (or simply Cilia extension) so as to allow Cilia mediators (mediation components) to be self-managed using the Cube framework. In this initial prototype version, we have only concentrated on ensuring basic management operations like self-creating, self-instantiating, and self-healing of mediators. The core modelled elements - Component, Node, and Scope - are perfectly suited for representing distributed Cilia mediation chains. For now, we have only implemented Monitoring/Executing components specific to the Cilia runtime. Moreover, we suppose that the Cilia chain runs in the same OSGi platform as the Cube Runtime. Otherwise, the implemented Monitor/Executor for Cilia should be enhanced with the capability to connect to remote Cilia execution nodes.

Cilia framework exports an administration service within the OSGi platform. The developed Monitor/Executor extension point for Cilia uses this service to interact with the local part of the Cilia mediation chain. This administration service allows the dynamic adaptation of local mediation components. The main role of the Cilia-specific Monitor/Executor extension point is as follows:

- The Monitor is connected to the administration service of the Cilia framework, and subscribed as listener for being notified of any changes that occur in the Cilia platform. Whenever notified, the Monitor summaries the change in the Runtime Model of the Cube Autonomic Manager. The Archetype Resolver of that Autonomic Manager will be consequently notified and will try to resolve any problems caused by the change with respect to the Archetype goals.
- The Executor is also connected to the administration service of Cilia framework, but subscribed as a listener to the Runtime Model of Cube's Autonomic Manager. Hence, when Managed Element instances are validated in the Runtime Model the Executor applies the corresponding changes into the Cilia application. Therefore, the Executor is bale to create, remove or update Cilia mediation components, as needed.

Cilia mediation components can be of different types. Namely, in Cilia, we distinguish between Mediator components and Adapter components. Hence, in the Archetype, users should add properties to the core Component element in order to distinguish explicitly between these two types of mediation components. The Monitor/Executor extension point for Cilia reads this information in order to decide whether to create a Mediator or an Adapter instance.

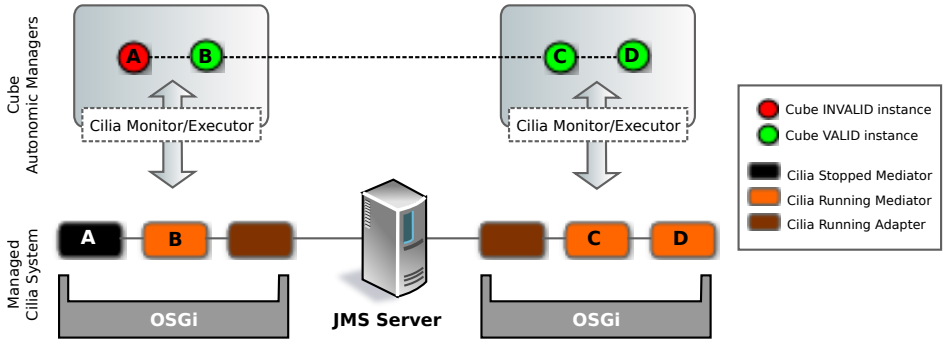


Figure 6.22: Cilia Monitors/Executors for the Cube framework

Figure 6.22 depicts a typical example of using the Cilia extension integrated with the Cube framework in order to self-manage Cilia mediation chains. This figure represents a snapshot of the application at a given time. The top level shows the Cube Autonomic Managers, only showing the Runtime Model content and the Cilia Monitor/Executor extension points. The bottom level shows the Cilia mediation chain deployed across two OSGi platforms. Here, Cilia uses a JMS server to exchange data between remote mediation components. Installing such mediation chain requires the instantiation of the Cilia components (mediators and adapters) and their appropriate configuration so that data messages can pass through the JMS server.

The first basic role of the Cilia extension is to start or stop Cilia mediation components depending on Cube's decisions. Whenever the Cube AM marks a Component instance as `INVALID` in the Runtime model, the Cilia extension stops the corresponding Mediator component in the real Cilia chain. Similarly, when this instance becomes `VALID`, the Cilia extension restarts the corresponding Mediator. When two Component instances at Cube level are connected (by the Archetype Resolver, in the Runtime Model),

the Cilia extension connects the two corresponding Mediation components at the Cilia level using Cilia Connectors. When the two components are located on the same Node, the Cilia extension sets in place a direct connector between the two (e.g., in Figure 6.22, between `A` and `B`, or `C` and `D`). However, when the two connected Components are located on different Nodes (e.g., between `B` and `C`), the Cilia extension generates JMS adapters on each side of the distributed mediation chain, and configures them so as to allow them to connect to the JMS server. The JMS server is required to transport data messages between distributed mediation nodes.

In future prototype versions, we plan to introduce Cilia-related concepts such as "Mediator", "Adapter" and "Connector" as modelled elements of the Cube framework. This will allow self-managing mediation chains based on more specific, low-level information from the mediation components.

The Cilia extension is identified by the "`fr.liglab.adele.cube.cilia`" namespace.

Graphical Monitoring Extension

This extension allows the graphical monitoring of the internal state of one local Runtime Model. It shows its constituents, which are modelled element instances, as a graph of connected vertices. The graph arcs represent references between these instances (see Figure 6.23).

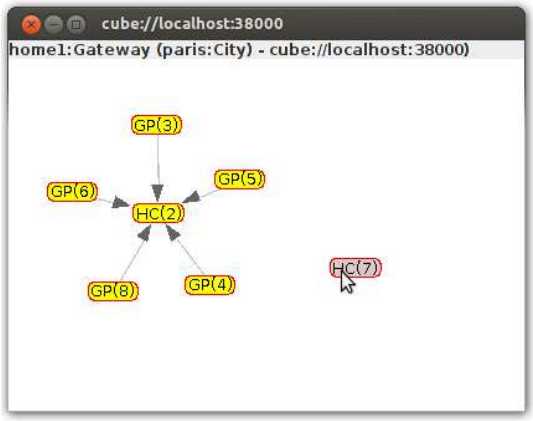


Figure 6.23: Graphical Monitoring of the Runtime Model of one Autonomic Manager

The only contribution of this extension is a Graphical Swing-based Java application provided as a Cube Executor. It reads the Autonomic Manager's Runtime Model dynamically and shows it using an Open Source drawing library - prefuse[8]. Any changes in the Runtime Model are reflected directly onto the visualization area of this graphical view. In future prototype versions we plan to allow the editing of Runtime Model instances directly via this graphical view tool. This will facilitate debugging and testing tasks. The Graphical Monitoring extension is identified by the "`fr.liglab.adele.cube.rm.monitoring`" namespace.

[8] http://prefuse.org

6.5 Related Tools

6.5.1 Overview

Cube related tools are OSGi applications that use the Administration Service provided by the Cube Runtime Bundle to handle and facilitate several tasks for users. In our prototype, we have implemented three different tools for different purposes:

- *Hot Deployer*: allows creating Cube's Autonomic Managers from a configuration file when it is placed in a per-defined directory;
- *Console Commands*: allows the debugging and management of local Autonomic Managers via simple console commands;
- *Dynamic Web Monitoring*: allows observing the global distributed Runtime Model in one place (web application).

In the following sub-sections, we provide some insights about each of these tools.

6.5.2 Hot Deployer

The *HotDeployer* is a directory-based deployment tool. It uses a directory in the file system (called 'load' in our implementation) to install and start an Autonomic Manager when its corresponding configuration file is placed there. It also dynamically deploys the extension bundles and other related bundles that are placed in that directory. The following Figure 6.24 depicts the architecture of the HotDeployer tool and its integration with the Cube Runtime component via the Administration Service.

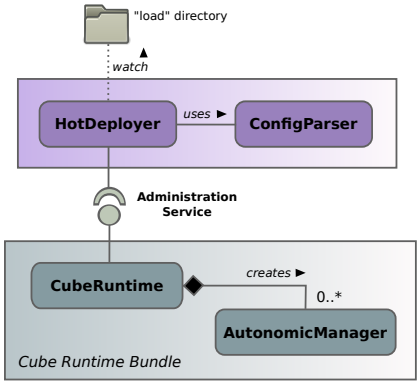


Figure 6.24: HotDeployer tool

To create an Autonomic Manager, HotDeployer needs a special XML configuration file, called Autonomic Manager Configuration File (amc files). When detected and parsed by the HotDeployer tool using the ConfigParser, This last one constructs its equivalent Java objects and calls the `createAutonomicManager` method of the Administration Service to create a new Autonomic Manager. When the configuration file is removed from the watched file system directory, its corresponding Autonomic Managers are automatically destroyed from the local OSGi platform.

Listing 6.13 gives an example of such configuration files.

Listing 6.13: Example of Autonomic Manager Configuration File

```
<cube cube-version="2.0">
  <autonomic-manager>
```

```
3       <host value="localhost"/>
4       <port value="38002"/>
5       <archetypeUrl value="file:../tutorial.arch"/>
6       <communicator value="fr.liglab.adele.cube.core:socket"/>
7       <auto-start value="true"/>
8       <rmcheck-interval value="2000"/>
9       <keepalive-interval value="2000"/>
10      <keepalive-retry value="1"/>
11      <debug value="true"/>
12      <perf value="false"/>
13      <persist value="false"/>
14
15      <extensions>
16        <extension id="fr.liglab.adele.cube.core"/>
17        <extension id="fr.liglab.adele.cube.osgi">
18          <property name="type" value="TABLET"/>
19        </extension>
20        <extension id="fr.liglab.adele.cube.rm.monitoring"/>
21      </extensions>
22    </autonomic-manager>
23  </cube>
```

Lines 3 to 13 provide configuration information for the Autonomic Manager itself. Lines 15 to 21 provide the list of extensions to be used by this Autonomic Manager. For each extension we should provide its identifier. We can provide a set of properties for each extension. Accepted properties are defined for each extension. Hence, the user should consult the extension's documentation for obtaining the list of accepted properties.

6.5.3 Console Commands

To allow developers to debug developed extensions or an already working system at runtime, we have implemented a command-based system. This tool is based on the Gogo shell tool provided by the Apache Felix project[9]. Users can type commands locally or remotely (via *telnet* for instance). Figure 6.25 shows the integration of this tool with the Cube Runtime via the Administration Service.

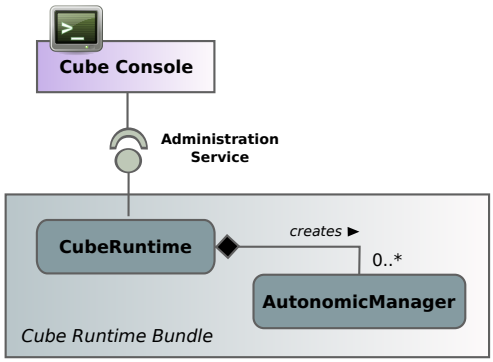


Figure 6.25: Cube Console

Appendix B.4 summaries the implemented commands and their syntax, and gives a short description about each one.

[9]http://felix.apache.org

Commands can be used with or without the “cube” namespace. Specifying the namespace explicitly is only needed when two commands from two different bundles have the same name.

For instance, Figure 6.26 shows a screen-shot of the result of the “rm” command.

```
Terminal - debbabi@Perillat: ~/dev/Cube/demo/tutorial2/server1
File  Edit  View  Terminal  Tabs  Help
g! rm 1
-----------------------------------------------------------------------
------ UNMANAGED ----
------ UNCHECKED ----
------ VALID --------
 + cube://localhost:38001/component/6ece1fd8-301f-4b4a-9f7b-65e56174fb75
    | PROPERTIES
    |   type=C
    | REFERENCES
    |   inputs:
    |     3380cfa1-0403-4054-8ea2-926b6126ff08
 + cube://localhost:38001/component/3020b757-461a-4156-ac75-8d838727ec4e
    | PROPERTIES
    |   type=A
    | REFERENCES
    |   outputs:
    |     3380cfa1-0403-4054-8ea2-926b6126ff08
 + cube://localhost:38001/component/3380cfa1-0403-4054-8ea2-926b6126ff08
    | PROPERTIES
    |   type=B
    | REFERENCES
    |   inputs:
    |     3020b757-461a-4156-ac75-8d838727ec4e
    |   outputs:
    |     6ece1fd8-301f-4b4a-9f7b-65e56174fb75
-----------------------------------------------------------------------
g!
```

Figure 6.26: Cube Console screen-shot

In this example, there are three created instances of type Component. Each instance has an UUID (6ece1fd8-301f-4b4a-9f7b-65e56174fb75 for the first one) which is also used to construct the URI of the instance regarding its current placement (which Autonomic Manager manages it). Each instance has a set of properties and a set of references. The “rm” command allows showing all this detail for all the local instances. For instance, we see that the third instance of type Component has an output reference to the first Component instance.

6.5.4 Dynamic Web Monitoring Console

Even if the aim of the Cube framework is to provide self-management behaviour for software systems, users need to monitor and to have a preview on what is happening in the system at any one time. The Web Monitoring tool contributes to achieving this objective by allowing users to visualize at runtime the global state of distributed applications.

To construct the global system state, the visualisation interface interacts with all the Cube Autonomic Managers in the system. The starting point is the Top Scope Leader (master) from which other AMs can be retrieved following their own Scope Leaders. Any changes occurring in an AM’s local Runtime Model are reflected dynamically in the global state diagram. This global model is represented as a visual graph, composed of modelled element instances and their relationships. The actual Cube prototype implementation only supports core, domain-specific, modelled elements - Components, Nodes, and Scopes. From the monitoring web interface, the user can retrieve information about any element of the system.

The Web Monitoring Console tool (Figure 6.27) is composed of three main parts:

- Front-end (web application)
- Aggregation Server
- Local Agent

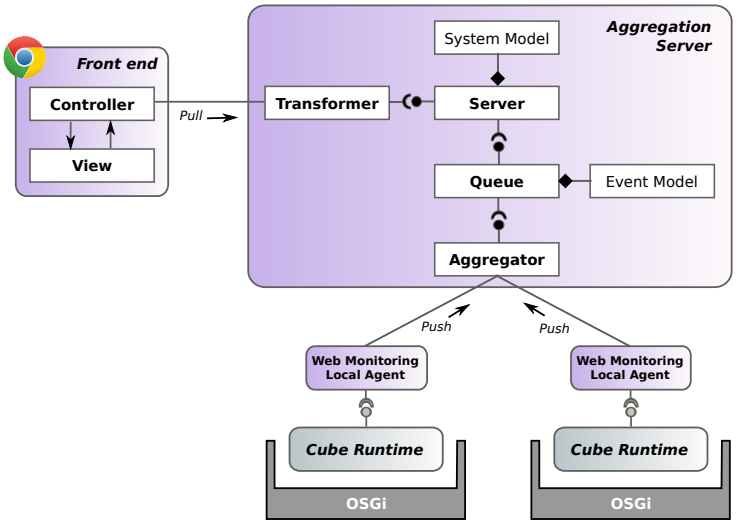


Figure 6.27: Dynamic Web Monitoring Console Architecture

The Local Agent is deployed directly onto the Cube Runtime OSGi platform. It allows monitoring the state of the local Runtime Model of Cube's Autonomic Managers. The monitoring state and dynamic changes are reported to the Aggregation Server using a push communication style. That is, information is pushed to the server from all the Cube Autonomic Managers available in the global system.

The Aggregation Server is responsible for collecting the data from all the Cube Runtimes. This data is pushed from the Local Web Monitoring Agent located on each OSGi platform. The Aggregation Server has a modular architecture composed of the following modules. The first module, "Aggregator", receives the data from the different agents. In the presented prototype, this is achieved using a simple Socket-based communication channel, where the Aggregator is a Socket server and the Local Web Monitoring Agents are Socket Clients. Hence, the Aggregator's role is to provide an interface with the monitored Cube Autonomic Managers; it does not perform any processing operations on the collected data.

The received data is forwarded to the next module, "Queue". The Queue accumulates and manages all the received data. This data is modelled as Events representing the different changes in the Cube system. The Queue keeps the order in which the change events have arrived and discards any duplicated events. At regular predefined intervals, the Queue sends the collected Events to the "Server" module, which updates the global System Model.

The System Model is an abstract, centralised representation of all the Cube's Runtime Model instances, at a given time. When started, the Server first interrogates all the Local Web Monitoring Agents to construct a first graph of the global system structure. Then, it regularly updates this system model when receiving a set of Events from the Queue representing the changes in the managed system. The last module of the Dynamic Web Monitoring System is the front-end web application. Its aim is to show the global application's state in a graph format.

We have developed an MVC-based[10] architecture for the web application. Here, the view is a JavaScript viewing component provided by JQuery's library - *arbor.js*[11] . This view component receives the model to display from the System Model of the Server module. This is ensured by an intermediate component, the Controller, which interrogates and pulls the Server module to get the System Model. At the

[10]Model-View-Controller Architecture
[11]http://arborjs.org/

Server level, another intermediate component ensures transforming the System Model (which is built using Java objects) to the adequate representation for the front-end interface - JSON objects in our case. For optimisation considerations, we only get recent changes from the System Model and only fragments of the Graphical View are refreshed when the System Model changes.

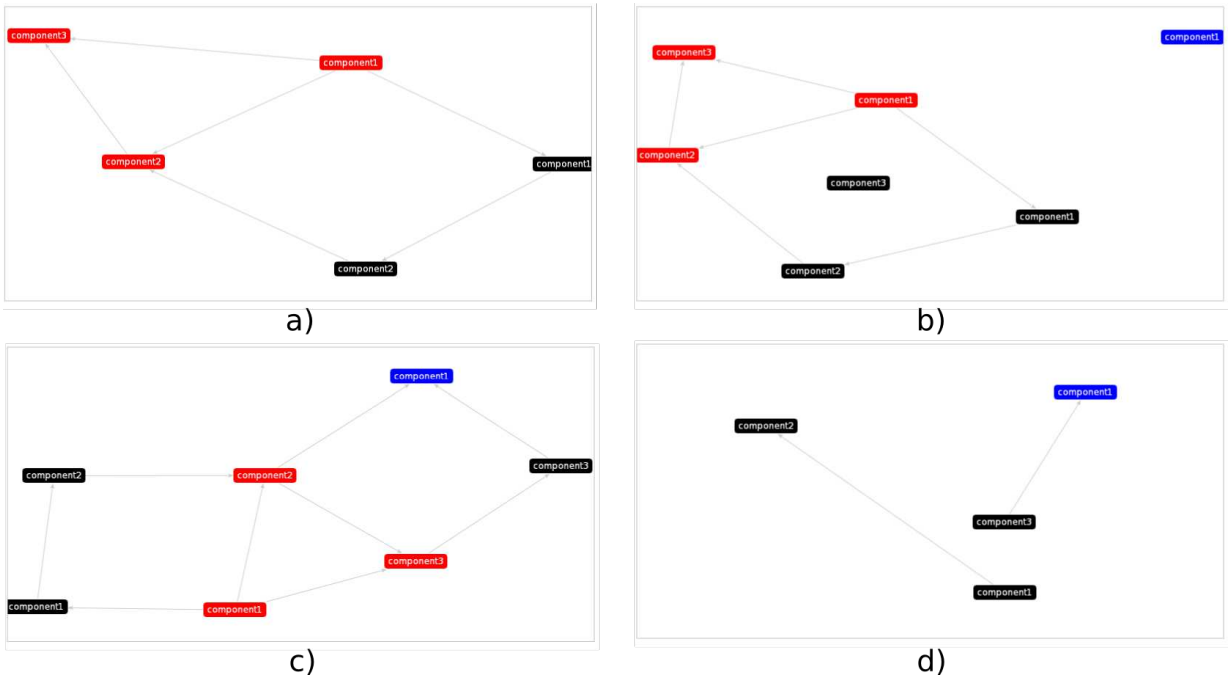


Figure 6.28: Dynamic Web Monitoring Console Example. a) initial model, b) two new components have arrived, c) new connections between the components, d) one node has departed

Figure 6.28 shows an example of the front-end interface when successive updates occur in the monitored Cube system. Components with the same colour are components hosted on the same node. In the figure 6.28, graph a) represents the initial state of the system. There are two nodes hosting five interconnected components (three components, in red, on one node and two components, in black, on the other node). In Graph b), a new component is created and placed on the black node – this change is reflected in the view interface. A new component located in a third node (blue) is also monitored. In Graph c), the last component located on the blue node is now connected to two components located in the black and the red node. In the last example in Graph d), the red node goes down; all its components are removed immediately from the graphical interface.

6.6 Summary

In this chapter we have presented the Cube framework and runtime prototype implementation, which is one of our main contributions to the Cube Project and to the self-management domain in general. The Cube framework and its main underlying principles were explained in the previous chapter. In this chapter we have detailed the implementation choices and explained the main Java classes of the implemented prototype.

We started this chapter by providing a small overview of the supporting technologies for the development and running of Cube Autonomic Managers. Here, we indicated that the Cube prototype is implemented using the Apache Felix iPOJO dynamic component model and runs on an OSGi execution platform. Combining these two technologies provided us with an innovative execution environment characterised by its high dynamicity and adaptation capabilities.

The Archetype specification for writing administration goals for the Cube framework was detailed in the previous chapter as an abstract model rather than as a concrete formal syntax (i.e. only based on graphical notation). In this chapter, we have proposed a formal XML syntax for the Archetype specification. XML is an universal and extensible language. We have explained how to write Archetype administration goals using XML. This XML file is then parsed and used by Cube Autonomic Managers.

The Execution Platform of the Cube system is composed of a set of OSGi platforms. Cube is packaged as an OSGi bundle which has a mandatory dependency with the iPOJO runtime bundle. The Cube bundle is in turn used by other bundles to implement related components. In particular, there are two kinds of related bundles: those providing extensions to Cube Autonomic Managers; and those providing Cube development and administration-related tools.

We have detailed the extension mechanism of Cube Autonomic Managers. There are four types of extension points via which users can introduce their contributions into a Cube AM. In particular, this allows specializing a Cube AM so as to be able handle the specific technologies of the targeted managed system.

Finally, we have outlined some of the developed tools. These range from local tools for a single AM, like the Graphical Monitoring tool of the local Runtime Model instances, to large-scale environments like the Dynamic Web Monitoring tool for visualising all the distributed Runtime Model parts from all the decentralized Cube Autonomic Managers.

Chapter 7

Evaluation

Contents

7.1 Overview . . . 148
- 7.1.1 Introduction to Mediation . . . 148
- 7.1.2 Cilia Mediation Framework . . . 149

7.2 Case Study 1: monitoring the consumption of home resources . . . 152
- 7.2.1 Context . . . 152
- 7.2.2 Managed System . . . 152
- 7.2.3 Problems addressed . . . 154
- 7.2.4 Solution proposed . . . 155
- 7.2.5 Scenarios and Results . . . 159

7.3 Case Study 2: health-care monitoring system . . . 167
- 7.3.1 Context . . . 167
- 7.3.2 Managed System . . . 168
- 7.3.3 Problems and Requirements to address . . . 169
- 7.3.4 Solution proposed . . . 170
- 7.3.5 Scenarios and results . . . 178

7.4 Evaluation . . . 181

7.5 Summary . . . 185

In chapter 4 we introduced the context, the problem, the objectives and the contribution of our thesis. In chapter 5 we presented the proposed Cube framework and in chapter 6 we detailed its implementation. In this chapter we evaluate our framework and its prototype implementation. The main objective of the evaluation process is to show Cube framework's viability for managing distributed mediation systems (i.e. via the Cube Archetype specification and Cube Runtime). To achieve this goal, two case studies have been developed utilising Cube autonomic features. These cases studies are: "*monitoring the consumption of home resources*" and "*health-care monitoring system*". These applications are parts of two different national research projects involving our research team . The first application represents the evaluation from a quantitative perspective. An experiment is set up to measure the required times for self-creation and self-configuration of a distributed application using the Cube framework. This is compared to performing the same tasks with and without autonomic capabilities. The second application represents the evaluation from qualitative perspective. We show how Cube is used to ensure some quality attributes like availability,

scalability and robustness by providing load-balancing and fault-tolerance features. We conclude this chapter via a discussion of Cube framework's qualitative attributes.

7.1 Overview

Cube framework was evaluated via two experimental case studies. The first case study (section 7.2) addresses the self-creation and configuration of a distributed component-based application (i.e. a data-mediation system). We show how Cube can be used to automate the creation and interconnection of component instances in distributed software systems. We also evaluate the performance and potential for scalability of our Cube solution when ensuring such objectives, in comparison with manual or centralized solutions. In the second case study (section 7.3), we focus the evaluation process onto additional, complementary autonomic features of the Cube framework. In particular, we show how the Cube framework can ensure the self-healing of distributed applications when the underlying execution nodes are subject to run-time crashes. We also show how Cube can self-optimise resource consumption across execution servers by load-balancing the processing tasks among a cluster's nodes. Hence, this second use case focuses more on the qualitative analysis of our framework.

The two evaluation case studies target the management of data mediation applications. Therefore, in the following sub-sections, we start by a short introduction to mediation systems (section 7.1.1) and introduce the used mediation framework: Cilia (section 7.1.2).

7.1.1 Introduction to Mediation

Mediation has been used historically to integrate data stored in IT resources such as databases, knowledge bases, file systems, digital libraries or electronic mail systems [Wie92, WG97]. In general, data mediation is implemented as a software layer between data sources (or resources) and data sinks (or applications) (see figure 7.1). This enables the interoperability and integration of disparate information-oriented elements in a timely fashion.

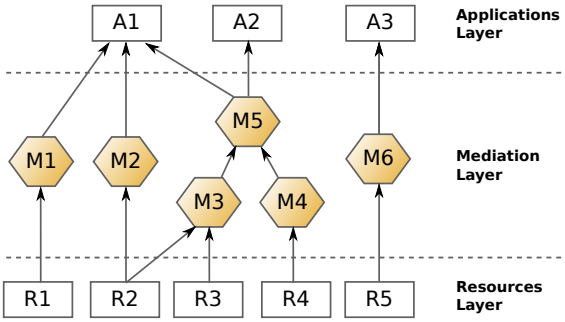


Figure 7.1: Mediation system

The mediation layer is composed of a set of mediators. G.Wiederhold defines a mediator as "*a software module that exploits encoded knowledge about certain sets or subsets of data to create information for a higher layer of applications*" [WG97]. Therefore, the principal objective of the mediation layer is to provide the interested applications with a high-level representation of the data obtained from the data sources. In general, such 'higher-level' representation will be based on concepts that are closer to the

targeted application's domain than to the 'lower-level' concepts of the data sources domain. To fulfil these objectives, mediators provide operations such as:

- Selection and filtering of incoming data depending on its characteristics.
- Conversion between incompatible data types.
- Aggregation of data obtained from multiple sources.
- Semantic alignment of inconsistent data.

Encapsulating such operations in dedicated data-mediation software is clearly a good practice. For instance, mediation software provides a single point of interface to the various applications and resources to be integrated. This reduces the number of connections needed and facilitates change management. In addition, mediation software provides an isolation layer from software details such as specific interfaces and data formats. Also, if configured correctly, it permits the quick and cost-effective development of new information services (e.g. by reintegrating data sources so as to produce new higher-level information). Moreover, by decoupling data sources from sinks, the mediation layer improves re-usability and evolution of both applications and data resources. It also permits the transparent addition of new QoS (Quality of Service) properties such as security, reliability, and so on. Finally, it can help improve the scalability of the entire system.

Nowadays, the scope of mediation-based approaches spans many applicability contexts, brought about by the emergence of new environments, new technologies and new applications. From the wide diversity of such contexts, two basic forms of mediation can be identified most often.

Data Integration: Mediators gather data from devices; apply operations on the collected data, including aggregation, filtering and correlation; and bring data to business applications or supervision tools, in a coherent manner. Since new data sources can appear and disappear at any moment, the mediation system must be sufficiently flexible and dynamic to undergo runtime adaptations without stopping the entire system. This mediation type is also known as Data Mediation.

Application Interoperability: Mediators deal with interoperation issues between heterogeneous software applications. In this context, the mediation software stands between client applications and provider applications (generally, Web Services). Its purpose is to enable service consumers to access provided services in a seamless and correct manner. Also known as Service Mediation, this form of mediation aims to provide a uniform or standardised interface to heterogeneous services. It was initially advanced by the ESB (Enterprise Service Bus) community as an essential part of ESB middleware [HTL05, HTL07]. In general cases, Service Mediation allows a Service Requester to connect to an appropriate Service Provider without worrying about specific interfaces and data heterogeneity.

7.1.2 Cilia Mediation Framework

Cilia [Deb09, GPD+10, GMD+11] is a lightweight and modular data-mediation framework based on OSGi technology and iPOJO service component-based model. The Adele Team has been the main developer of this framework, which is currently being used in collaborative initiatives like the Medical project (see use case 2, section 7.3). Cilia proposes both a specialized component model for data-mediation systems, and a dynamic execution environment. The adoption of a component model aims to enable the construction of mediation applications by means of composition of reusable components, hence ensuring better application modularity and flexibility. Indeed, in addition to modularity, dynamicity is an important property of the Cilia framework, endowing mediation applications with sufficient flexibility for changing their architecture at run-time.

Technically speaking, a Cilia mediation application (also called mediation chain) is a set of mediation components - called Mediators - that are interconnected by links – called Bindings. Hence, such mediation applications take the form of a directed acyclic graph (DAG), where nodes are Mediators (receiving, processing and forwarding data) and edges are Bindings between Mediators (transmitting data).

A Cilia Mediator has the following internal sub-components (Figure 7.2):

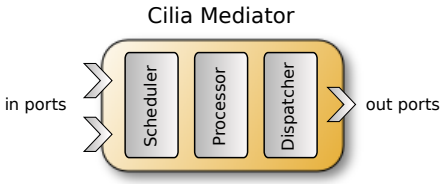


Figure 7.2: Cilia Mediator component and its constituents

Scheduler: responsible for collecting input data from one or several sources until a pre-defined condition is met. When this happens, the scheduler forwards all data collected so far to the mediator's processor (discussed next). Each scheduler implementation provides its own logic for the associated triggering condition. This may be temporal, quantitative, qualitative and so on. Some typical examples of scheduler implementations include: "Periodic Scheduler", which sends collected data periodically to the processor; and, "Immediate Scheduler", which triggers the processing as soon as the data arrives. Cilia distribution provides several predefined schedulers. Cilia users can develop additional schedulers or configure the existing ones.

Processor: responsible for processing the data received from the scheduler and forwarding it to the dispatcher (discussed next). Some illustrative examples of processors include: "String Splitter", which splits data based on a regular expression; or, "XSLT Transformer", which receives data in the form of XML documents and applies an XSLT transformation to them. Developers can implement their specific processors via Java classes that do not have to implement any framework-related interfaces. To include such custom processor classes into Cilia, developers must additionally provide a declarative XML specification file, with information about the processor's class name, the processing method's name and the type of data that it can receive.

Dispatcher: responsible for dispatching the processed data received from the processors to one or several destinations. The most common examples of dispatcher implementations include: "Multi-cast Dispatcher", which sends a copy of the processed data to all destinations (e.g. connected mediators); "Content-Based Dispatcher", which sends the processed data to a particular destination for which some condition related to the data content is met. Cilia distribution provides several predefined dispatchers. Cilia users can develop additional dispatchers or configure the existing ones.

In addition to the mediators described above, Cilia provides a special kind of mediators called *Adapters*. They are placed at the boundaries (input/output) of Cilia mediation chains, in order to interface them with heterogeneous (non-Cilia) data sources and sinks, respectively. Hence, adapters are responsible for the communication with external data sources (resources) or sinks (target applications). There are three types of adapters: in adapters, out adapters, and request-response adapters.

Mediators (and Adapters) are connected via Bindings. A binding describes a connection between an output and an input port. Connected mediators form a *mediation chain*. Figure 7.3 illustrates a sample mediation chain composed of two adapters, four mediators and six bindings.

Cilia provides a reflective API [GMD+11] which allows creating, updating and removing mediators

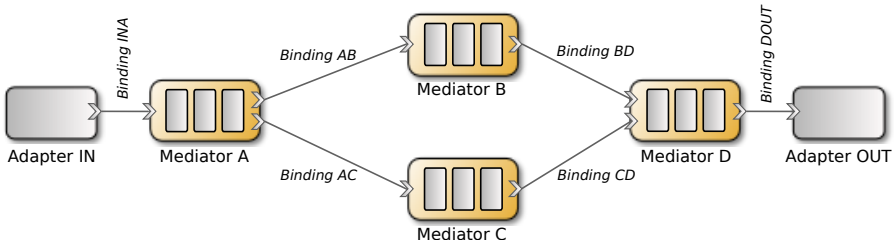


Figure 7.3: Example of Cilia mediation chain

and bindings from a mediation chain dynamically without stopping the application. This API is exported as a local OSGi service and can be used by third party tools to manipulate mediation chains. We use this API to implement Cube framework's extension for Cilia, as explained farther in this chapter.

Cilia chains are composed exclusively of local mediators (i.e. located on the same OSGi platform). To create a distributed mediation chain, one must introduce specific Adapters for connecting several local mediation chains together. Hence, Adapters provide the remote communication function for distributed mediation chains. They can be implemented to support various communication protocol(s) or middleware(s). This approach helps separate the chain's data processing functions from its data transport functions. Nonetheless, creating and connecting local mediation chains to form complete distributed mediation applications requires a certain amount of expertise, in both Cilia and the communication technologies employed. This task becomes rapidly cumbersome when the mediators are distributed over hundreds or thousands of execution nodes dispersed in separated administrative domains and/or geographical areas.

For local mediation chains, Cilia provides a relatively simple mechanism for creating such chains without using the reflective API. Instead, users can write specific XML files to describe the local mediation chain in terms of the set of Mediators, Adapters and Bindings between them. This special-purpose XML file is called DSCILIA. Listing 7.1 depicts an example of a local mediation chain defined based on DSCILIA syntax.

Listing 7.1: Example of DSCILIA XML file

```
<cilia>
  <chain id="Test-Chain">
  <!-- Adapters instances definition -->
  <adapters>
    <adapter-instance type="PeriodicPresenceDetectorAdapter" id="presenceDectector">
      <property name="delay" value="500" />
      <property name="period" value="1000" />
    </adapter-instance>
    <adapter-instance type="console-adapter" id="exitAdapter" />
  </adapters>
  <!-- Mediators instances definition -->
  <mediators>
    <mediator-instance type="HelloMediator" id="hello">
      <ports> <in-port name="in"/> <out-port name="out"/> </ports>
    </mediator-instance>
  </mediators>
  <!-- Bindings definition -->
  <bindings>
    <binding from="presenceDectector" to="hello:in" />
    <binding from="hello:out" to="exitAdapter" />
  </bindings>
```

```
22   </chain>
23 </cilia>
```

This XML file was specified for creating a mediation chain on a single gateway. The mediation chain is composed of one mediator (HelloMediator – lines 15-20) and two adapters (PeriodicPresenceDetectorAdapter input adapter – lines 5-9, and Console output adapter – line 10). Data collected using the PeriodicPresenceDetectorAdapter is sent to the HelloMediator (first binding - line 25). In this simple example, the mediator simply forwards data received to the output adapter (second binding – line 26), which displays the data in the terminal.

7.2 Case Study 1: monitoring the consumption of home resources

7.2.1 Context

In this use case, we consider the autonomic management of a distributed data-mediation system for monitoring the consumption of home resources, including electricity, gas, and water. Generally, the purpose of data-mediation applications is to collect data from several sources, then transport and process the data so that it can be consumed by several sinks. The use case considers multiple data sources representing resource monitoring probes and one sink representing a global cost calculator for consumed resources. The data-mediation application consists of a number of interconnected Mediators. Each Mediator processes data from several sources, or from other Mediators, in order to calculate consumption costs at various granularity levels, including home, city and country (see next sub-section 7.2.2).

In this use case, we use Cube to automatically create and maintain a data-mediation application that meets predefined architectural goals (constraints), specified formally in the administrator's Archetype. The primary goals are about the structure of the global mediation chain, in particular about mediator types and the way they are connected at different levels of the distributed mediation application.

This use case was part of the the *Self-XL* project[1] funded by the French institute "Agence Nationale de Recherche ANR" (from 2009 to 2012). This project sought to acquire new processes in the field of autonomic computing, by identifying and investigating the components (e.g. languages and runtime) that would facilitate the implementation of complex and large-scale autonomic systems. The scope of this project encompasses any computing system featuring high software complexity, in terms of its scale, distribution, dynamism, heterogeneity, and so on. Targeted systems range from cluster computing to pervasive embedded systems, including recent and legacy software.

The first prototype of the Cube framework was developed during this research project. An elaborated use case was specified to evaluate the framework and the corresponding prototype. The initial use case was later re-evaluated for new versions of the framework and published at SASO'12 international conference [DDL12]. In this section, we present the same use case as specified in the Self-XL project. However, we provide a new evaluation based on results obtained from experiments with the most recent prototype version.

7.2.2 Managed System

As an application example, we consider a mediation system for monitoring the consumption of home resources. The sample system monitors the consumption of household resources, including electricity, gas, and water. Collected data is processed for calculating different costs. At the house level house, it calculates electricity, gas and water costs. These are then aggregated into higher-level region costs, city

[1] http://selfxl.forge.inria.fr

costs and then national costs. The corresponding Component types include specific probes – collecting electricity, water, and gas measurements; and various cost calculators – computing consumption costs at the house, region, city and national levels.

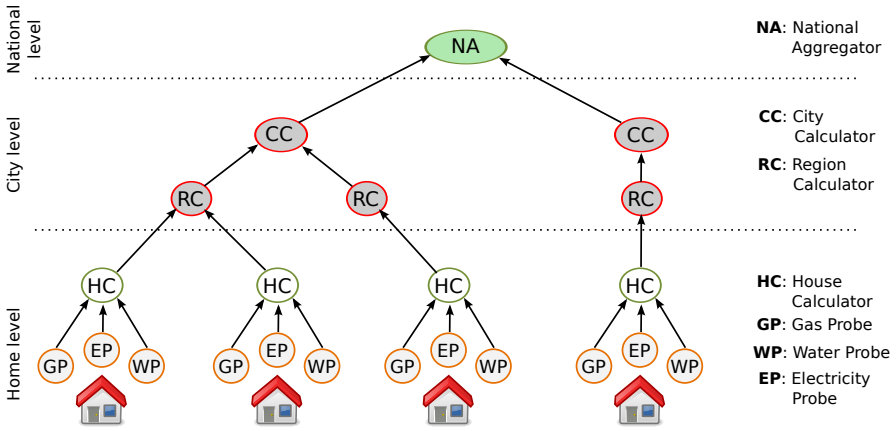


Figure 7.4: Overview of a house resource monitoring application

Figure 7.3 shows the overall architecture of our distributed mediation system. We differentiate three levels in which different kinds of components exist. These levels were identified based on the types of administrative authorities involved and the corresponding underlying hardware platforms.

At the bottom level, representing the houses, three kinds of components are introduced to collect gas, electricity, and water measurements. We call them Gas Probe (GP), Electricity Probe (EP), and Water Probe (WP). These components execute on a special kind of machines called gateways (or home boxes), available in each house. Each of these probe components sends the collected data to a House Calculator (HC) component, which aggregates the data, calculates the overall house consumption cost and forwards it to a regional cost calculator (discussed below).

At this intermediate level, representing entire cities, two kinds of components exist. The first one, called Region Calculator (RC), collects data from the set of houses located within one region of the city. Hence, for each region we have one instance of this component. All RG instances in a city send the collected data to a unique instance at the city level, called City Calculator (CC). In turn, each CC instance analyses the collected data, summaries it, and sends it to a national data-centre (discussed next). At the city level, we have interconnected server machines. These machines can be located in a single cluster, or distributed across various locations (e.g. geographically close to monitored houses).

Finally, at the topmost national level, a single mediation component instance of type National Aggregator (NA) receives and processes data from the different City Calculators. In such scenario, a powerful server cluster should, in principle, be set in place at this level to support the load of the national data-centre (for both data aggregation and dissemination to numerous clients).

For reasons of performance and of distribution of infrastructure-related costs among the various authorities concerned (i.e. house, city and national authorities), in this use case, administrators wanted to limit the number of inputs of some components. In particular, the RC component instance is limited to receiving a maximum of fixed number of input connections from HC components. Lastly, an additional constraint was required to impose having a single CC component instance within all the servers that belong to one city.

7.2.3 Problems addressed

The primary objective of this first case study is to evaluate Cube framework's capability to self-create (i.e. instantiate and connect components) distributed mediation chains that conform to an archetype specification.. To confirm this capability, Cilia mediators should be instantiated automatically and their interconnections configured correctly, via the Cube framework. At the local level (within one OSGi platform), bindings between Cilia mediators are 'direct' - using either OSGi's local Event Admin service, or direct method calls. However, when crossing the boundaries of the local OSGi platform, administrators use Cilia JMS Adapters to transmit data messages via a common JMS[2] Server (as depicted in Figure 7.5). Note that the use of JMS Adapters represents a specific technological choice in the presented use case. Other middleware technologies may also be adopted for interconnecting distributed Cilia chains.

The default activity necessary for instantiating Cilia mediation chains is to write a DSCilia XML file for each local mediation chain to be executed on each OSGi platform. When a DSCilia file is deployed in a special-purpose directory, the "*static*" mediation chain specified in the DSCilia file is instantiated automatically (by Cilia). To connect several local mediation chains and form a global distributed mediation chain, administrators should create adequate JMS Adapters in each OSGi platform. These JMS Adapters should be configured to use an available JMS Server and to channel the data towards the targeted mediation chain(s).

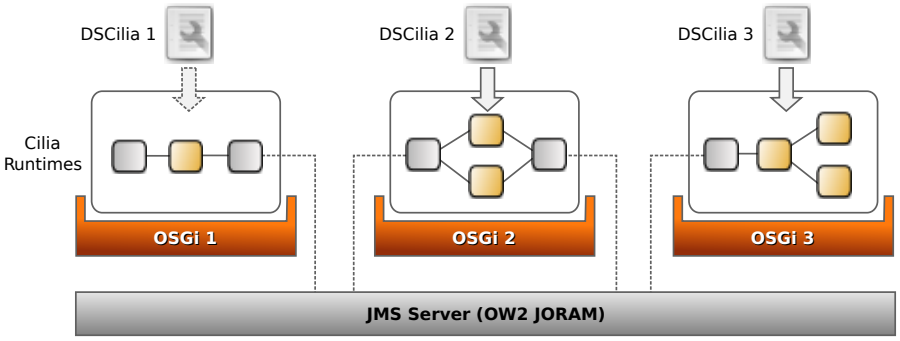


Figure 7.5: Distributed Cilia mediation chain

This operation is relatively simple because of Cilia's dynamic adaptability features. However, configuring all the local mediation chains located across (very) large areas can quickly become costly and prone to human error. In particular:

- Administrator should maintain hundreds or thousands of local DSCilia files.
- Changing one Mediator or Adapter in one DSCilia (local mediation chain) can have serious repercussions on other mediation chains.
- Migrating some mediators to other execution nodes can require complicated reconfiguration tasks. This is mainly because DSCilia files are intended primarily for specifying and creating static local mediation chains.
- In large-scale distributed mediation systems, it becomes highly difficult for administrators to have a global view of all the mediation chains and of their interconnections.

[2]Java Messaging Service

7.2.4 Solution proposed

In this section we detail how we used the Cube framework to tackle the aforementioned problems.

As we have outlined in section 4.4.5 of chapter 4, our approach to self-management using Cube involves several steps and activities. The first two activities are performed manually and off-line. They involve application-specific knowledge extraction and specification. In the context of our proposed framework, these correspond to: selecting or defining new model abstractions; and, selecting or developing new extensions, which provide either specific monitor/executor components and/or specific resolvers related to the managed system. Next, users should define administration objectives based on the Archetype specification language. The resulting Archetype file is then used by Cube Autonomic Managers to self-manage the system and ensure that user goals are met. We detail each of these steps for this use case in the following sub-sections.

7.2.4.1 Model Abstractions

For this first use case, we will only use model abstractions provided by Cube framework's Core Extension; these suffice to address our requirements. Therefore, as outlined in the previous subsection (7.2.3), the problems addressed concern mainly the instantiation and connection of distributed components. Hence, details about the internal components, behaviour, data, or performance are beyond the scope of our evaluation in this case study. Table 7.1 summaries the core model abstractions used and their semantics with respect to our distributed mediation chain application.

Table 7.1: Case study 1: model abstractions and their semantics

Model Element	Semantic
Component	Used as an abstraction for a Cilia Mediator or Adapter. We need the following property in particular: • **kind**: used for differentiating between the two concepts - Mediator and Adapter. Its value indicates the nature of the Cilia component - "M" for Mediator and "A" for Adapter.
Node	Used as an abstraction for an OSGi platform. Each OSGi platform contains a local mediation chain with a set of Mediators (and Adapters). Each Cube Autonomic Manager controls a single Node instance.
Scope	Used to regroup a set of OSGi platforms, in order to simplify the description and resolution of goals that span over those platforms. For instance, we use the Scope model abstraction to regroup all the OSGi platforms that belong to the same city.
Master	The corresponding model instance will contain the list of all scope leaders. This is necessary for implementing the federation-like organization of Autonomic Managers, which was the design option selected for the presented Cube framework.

7.2.4.2 Extensions

To use the model abstractions detailed above, we must add the Core Extension to all Cube Autonomic Managers participating in our use case. The Core Extension provides these model abstractions, as well as a set of corresponding Specific Resolvers for dealing with the goals and properties that can be defined in the

Archetype based on these abstractions. We can reuse these provided artefacts in this use case. However, we need to implement an additional extension for this use case - a new monitor/executor extension for handling Cilia mediation chains. Table 7.2 summaries the list of extensions used (reused or implemented) and their configuration properties.

Table 7.2: Case study 1: used extensions and their possible configuration properties

Extension Name	Extension Namespace	Configuration Properties
Core Extension	fr.liglab.adele.cube.core	master
Cilia Extension	fr.liglab.adele.cube.cilia	jms.server
		jms.port
Script Extension	fr.liglab.adele.cube.script	

Core Extension

The Core Extension is provided by the Cube framework. In this use case, we reused some of the Core Extension's model abstractions and a set of Description and Goal Properties. Examples of the Properties we reused include: *hasComponentType*, *hasNodeType*, *hasScopeId*, *onNode*, *inScope* and *connected*.

The Core Extension has a configuration property called "master", which, if set to "true", triggers the instantiation of a Master element. Consequently, the Autonomic Manager that contains the Core Extension configured this way (i.e. master = true) is considered to be the master node. The Master AM instance holds the list of all scope leaders (their URLs). In the Archetype, we should explicitly mention the address of this master node, since it represents the entry point for all Autonomic Managers participating in Cube's self-management activity.

Cilia Extension

We have implemented this extension specifically for this use case, in order to enable Cube to administer (monitor and act upon) Cilia mediation chains. The extension consists of a Cilia Monitor (for observing Cilia mediation chains) and a Cilia Executor (for acting upon Cilia mediation chains). These were further detailed in the previous chapter 6 (subsection 6.4.4).

In this first use case, we aim to have the Cube framework exercise full control over the distributed Cilia mediation chain. This implies that all the Cilia mediators and bindings are created by Cube. In this context, Cube's Cilia Monitor plays a less important role than Cube's Cilia Executor. This is because the Executor is responsible for translating the modelled element instances (in Cube's Component Runtime Model) to their equivalent Cilia Mediators (in the actual mediation chain). Moreover, the Cilia Executor is responsible for creating the various bindings between mediators, and, if a remote reference is present in the Runtime Model, for creating the adequate Cilia JMS Adapters that connect the remote Mediators via JMS (as described in the previous chapter 6, section 6.4.4).

Finally, the Cilia Extension provides some Specific Resolvers for handling Cilia-related concepts. In particular, we provide the *hasScheduler* and *hasDispatcher* Properties to allow administrators to specify which kind of schedulers and dispatchers should be used for particular Cilia Mediators.

Script Extension

The Script Extension helps automate some of the particular set-up procedures for this use case. These consist in initializing several Autonomic Managers with pre-created instances. Most importantly, the

Extension Script allows the manual creation of Node instances corresponding to each managed OSGi platform, via relatively simple script commands. For each Autonomic Manager, the administrator should configure the Script Extension so as to create the appropriate Node instance. Listing 7.2 shows an example of using this Script Extension.

Listing 7.2: Simple script file for creating Cube Runtime Model instances

```
1 <extension id="fr.liglab.adele.cube.script">
2   <property name="1" value="newi Node type=Gateway"/>
3   <property name="2" value="newi Component type=HC"/>
4 </extension>
```

As shown in the example, script commands are provided as extension properties where the name of the property indicates the execution sequence of the commands, and the value of the property defines the actual command to execute. For instance, the "newi" command in the example allows the creation of modelled element instances in the Runtime Model. Here, the nature of the created instance is defined via the first parameter and the instance's initial properties via the second parameter. Hence, in line 2, we start by creating the Node instance of type "Gateway", and in line 3, we create a Component instance of type "HC".

Finally, Listing 7.3 shows a complete example of a configuration file for one Autonomic Manager using all the three presented extensions.

Listing 7.3: Configuration file for one Cube's Autonomic Manager

```
1  <cube cube-version="2.0">
2    <autonomic-manager>
3      <host value="192.168.0.10"/>
4      <port value="38002"/>
5      <archetypeUrl value="file:../uc1.arch"/>
6
7      <extensions>
8        <extension id="fr.liglab.adele.cube.core">
9          <property name="master" value="true"/>
10       </extension>
11       <extension id="fr.liglab.adele.cube.cilia">
12         <property name="jms.server" value="192.168.0.100"/>
13         <property name="jms.port" value="16010"/>
14       <extension>
15       <extension id="fr.liglab.adele.cube.script">
16         <property name="1" value="newi Node type=Gateway"/>
17       </extension>
18     </extensions>
19
20   </autonomic-manager>
21 </cube>
```

The Autonomic Manager is specified to have "`192.168.0.10`" as host and "`38002`" as port. Its Archetype is located at the destination indicated in line 5. Three extensions are declared: the first one is the "Core Extension" configured to be a master; the second one is the "Cilia Extension", where the information on the JMS server is provided in lines 12 and 13; and finally, the "Script Extension" with one command allowing the creation of one Node instance of type "Gateway".

7.2.4.3 Archetype

For clarity and simplification, we have divided the Archetype specification into three parts: (1) nodes self-initialization, (2) self-creating and connecting home mediators to the city servers; and (3) self-creating and connecting city mediators to the national mediator. The entire Archetype XML file is listed in Appendix C.

The first part (depicted in Figure 7.6), shows the goals related to the self-initialization of the mediation system's Nodes (see section 5.3 of chapter 5 for a description of this notation). The actions required by these goals include initialising the different Nodes and grouping them dynamically into predefined Scopes. They also include the instantiation of the first mediator component on each Node (or local platform). This, in turn, will fire the self-instantiation of the remaining mediation chain.

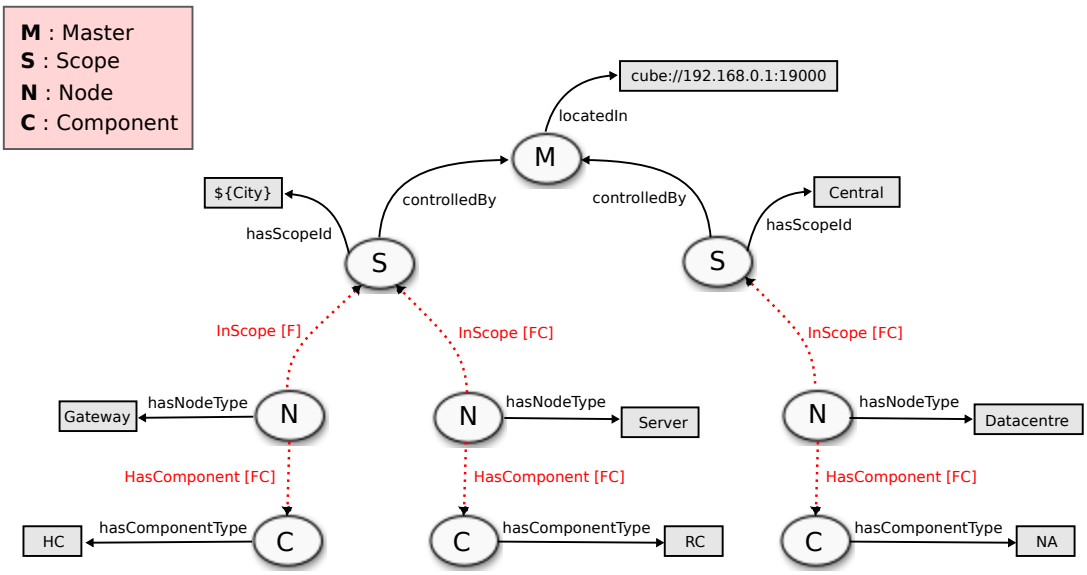


Figure 7.6: Archetype part for Node self-management in Case Study 1

The exemplified Archetype defines three kinds of Nodes: Gateway, Server and Datacentre. The different Gateways and Servers should be regrouped within the same Scope - called City. This objective is specified using the "inScope" Goal Property (between the Nodes and the Scope). The "inScope" goal is tagged with the [F] resolution strategy for the Gateway nodes, but with [FC] resolution strategy for Servers. This means that the Autonomic Manager controlling the city server should find a scope leader or create a new scope leader instance (if not found); while the Autonomic Manager of the Gateway should only try to find a scope leader. In this setting, the Gateway manager cannot become a scope leader itself, for technical considerations such as limited resources on home boxes.

The City scope description features a "hasScopeId" Description Property, which is set to "${City}". This means that the actual scopeId value will depend, case-by-case - in this example, on the actual city name (e.g. 'Grenoble', 'Lyon' or 'Paris'). Hence, each Autonomic Manager controlling a Gateway or a Server node should look for a scope that has the same id as the Autonomic Manager's own city property. For instance, if we consider a Gateway located in the city of Grenoble, the Gateway's Autonomic Manager should be started with a property "city=Grenoble". Consequently, when the Autonomic Manager will try to resolve the Archetype, and will look for a suitable scope for its Gateway node, it will try to find a scope that has 'Grenoble' as identifier.

At the national level, Datacentre nodes should be regrouped within a Central scope. The Central scope and all the City scopes are controlled by a single Master instance located at “cube://192.168.0.1:19000”.

Also in the first Archetype part, we specify that the Autonomic Manager on each Node should instantiate the first mediator of the local mediation chain of that Node. For instance, on each Gateway we should create an “HC” Mediator, for calculating the cost of consumed resources within a home. We use the “HasComponent” Goal Property to ensure this objective.

The second part of the Archetype (depicted in Figure 7.7) shows the goals related to the self-creation and configuration of the mediation chain part that is located on home Gateway nodes. The main goals in this Archetype part define the way in which Components must be interconnected (using the “connected” Goal Property).

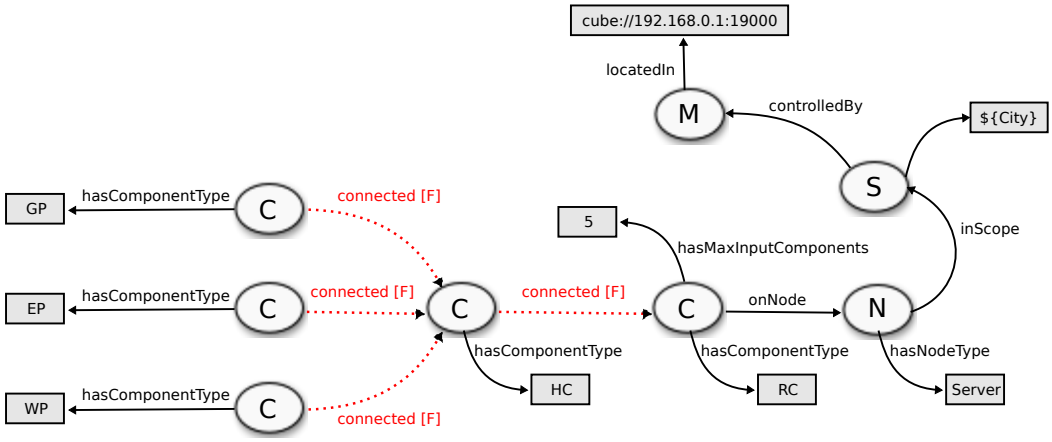


Figure 7.7: Archetype part for self-creating and connecting Mediation components on home Gateways in case study 1

Four Component Types are to be managed at the home level - Gas Probe (“GP”), Water Probe (“WP”), Electricity Probe (“EP”), and House Cost Calculator (“HC”). Instances of the first three Component Types, or Probes, should be connected to the “HC” Component instance. At the same time, the “HC” instance should be connected to a remote instance - “RC” (Region Calculator) - located on a Node of type Server in the City cope of that region. We have also specified that the found “RC” instance should not have more than five (“5”) input components; this was indicated via the “hasMaxInputComponents” Description Property on the “RC” Component. If this is not the case, Cube should find another “RC” instance on the same remote Node or on any other Node that matches the provided description (i.e. Node Type “Server” and in Scope “${City}”).

The third part of the Archetype (depicted in Figure 7.8) shows the goals related to the self-creation and configuration of mediation chain parts that are located on City Servers and National Datacentres. Here, all “RC” component instances should be connected to the “CC” (City Calculator) component instance that is located on one of the Servers in that City. In turn, any “CC” instance should be connected to the “NA” (National Aggregator) component instance located on a Datacentre.

7.2.5 Scenarios and Results

We executed several experiments on a distributed software platform (50 Cube runtimes) deployed on a single physical machine. The main purpose of these experiments was twofold. First, we aimed to

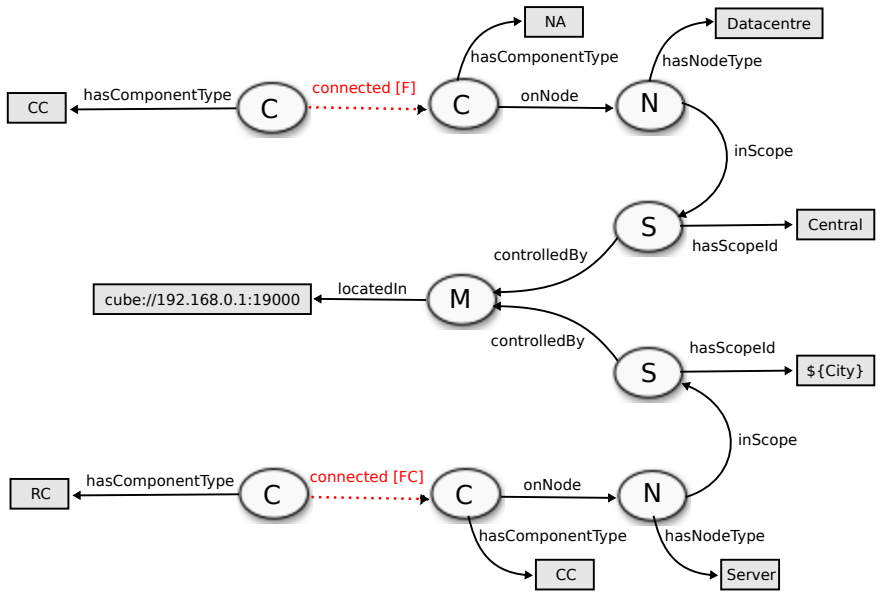


Figure 7.8: Archetype part for self-creating and connecting City and National Mediation components in case study 1

illustrate Cube framework's capability of creating and configuring distributed mediation chains that meet an archetype's goals. Second, we aimed to obtain several generic indicators on the Cube framework's performance, in order to predict the way in which the resolution process will scale with the number of managed platforms.

The experimental machine consisted of a laptop PC (Core 2 Duo 3.06 GHz and 4 Gb of RAM). This machine hosts the OSGi platform containing the Cube Runtime bundle and the different Cube Extension bundles. OSGi runs on an OpenJDK 1.6 JVM executing on a Linux Ubuntu OS. All Cube Autonomic Managers in the experiments are created on this machine. Each Autonomic Manager is executed as a separate thread and connects to other Autonomic Managers via a network stack (via TCP/IP). Hence, the Autonomic Managers could also be deployed on different physical machines and feature the same behaviour, only with higher delays.

We have evaluated the implemented Cube prototype via the configuration depicted in Table 7.3.

Table 7.3: Autonomic Managers for Case Study 1

AM n°	**Host**	**Port**	**Role**
1	`localhost`	`19000`	Master node
2	`localhost`	`19001`	National Datacentre
3-5	`localhost`	`19100-19102`	Grenoble Servers
6-10	`localhost`	`19200-19204`	Paris Servers
11-25	`localhost`	`19300-19314`	Grenoble Gateways
26-50	`localhost`	`19500-19424`	Paris Gateways

In total, we have a maximum of fifty Cube Autonomic Managers. We ran a series of experiments, where for each experiment we incremented the number of participating Autonomic Managers - from

eleven to fifty. The purpose was to test Cube's behaviour when the number of Autonomic Managers (and hence of managed sub-systems) increases.

Hence, in each individual experiment, we configure the distributed platform to include a certain number of managed platforms (with one AM per platform); and we observe the number of control messages exchanged between the Autonomic Managers, as well as the resolution time taken by each Autonomic Manager for ensuring the Archetype objectives.

The Autonomic Managers numbered from 1 to 10 are present in all the experiments, since they manage Nodes that belong to the system's infrastructure (Servers and Datacentre); we assume here that there is no variation at this level. On the contrary, we assume that the system may include various numbers of houses and we simulate this by incrementing the number of connected houses in each experiment, as described before.

As previously indicated, we are mainly interested in verifying Cube framework's capability of finding a correct instantiation solution to the Archetype; and in calculating some indicative performance metrics, such as the number of exchanged control messages and the resolution time of the individual Autonomic Managers. Hence, for the purpose of these experiments, we have not included the Cilia Extension in the Autonomic Managers; this would have simply shown Cube framework's capability of translating the Autonomic Managers' instantiation solution, expressed via their respective Runtime Models, into actual Cilia mediation chains.

We also wanted to make Cube framework's performance evaluation as independent as possible from the performance of the underlying distributed platform (e.g. processing and network communication delays). Since the autonomic management process is decentralised, the processing overhead on each Autonomic Manager will be limited and not increase significantly with the number of managed platforms – indeed, the Autonomic Managers will share the overall management load among them.

Most importantly, however, as with any decentralised management solution, the main scalability concerns are related to the communication overheads induced by the coordination procedures of independent Autonomic Managers. Therefore, we focused on determining the number of messages exchanged between Autonomic Managers during the resolution process, rather than on the exact time delays required by this process. This allows us to evaluate, in a qualitative fashion, the scalability of the proposed decentralised solution; whereas the exact performance characteristics will depend on the performance of each distributed deployment platform.

A total of 40 distinct experiments were carried-out, with one additional Gateway included in each experiment, with respect to the previous experiment. A maximum of 40 Gateways were used during these experiments, with 15 home Gateways located in the city of Grenoble (with AM indexes from 11 to 25) and 25 home Gateways in the city of Paris (with AM indexes from 26 to 50). The Gateways in Grenoble were added progressively over the first 15 experiments; then the Gateways in Paris were added over the remaining 25 experiments. In each experiment, the Cube Autonomic Managers had to start the resolution process from scratch, in order to instantiate a distributed mediation chain that met the archetype goals.

Figure 7.9 illustrates the manner in which the archetype defined for this case study (section 7.2.4.3) is being resolved, in three different experimental settings. The purpose of these examples is to show how the Cube framework manages to find instantiation solutions that conform to the archetype, while being adapted to each deployment platform. This is achieved via a decentralised archetype resolution process based on cooperating Autonomic Managers (one for each platform).

In the figure, the illustrated instances of distributed mediation chains correspond to the solutions found by archetype resolvers and stored in their corresponding Runtime Models (one on each platform). For simplicity, we only show the created Mediator instances and their interconnections; we ignore the additional modelled element instances that are also created in the Runtime Model during the resolution process (e.g.

we ignore modelled instances of Nodes and Scopes.

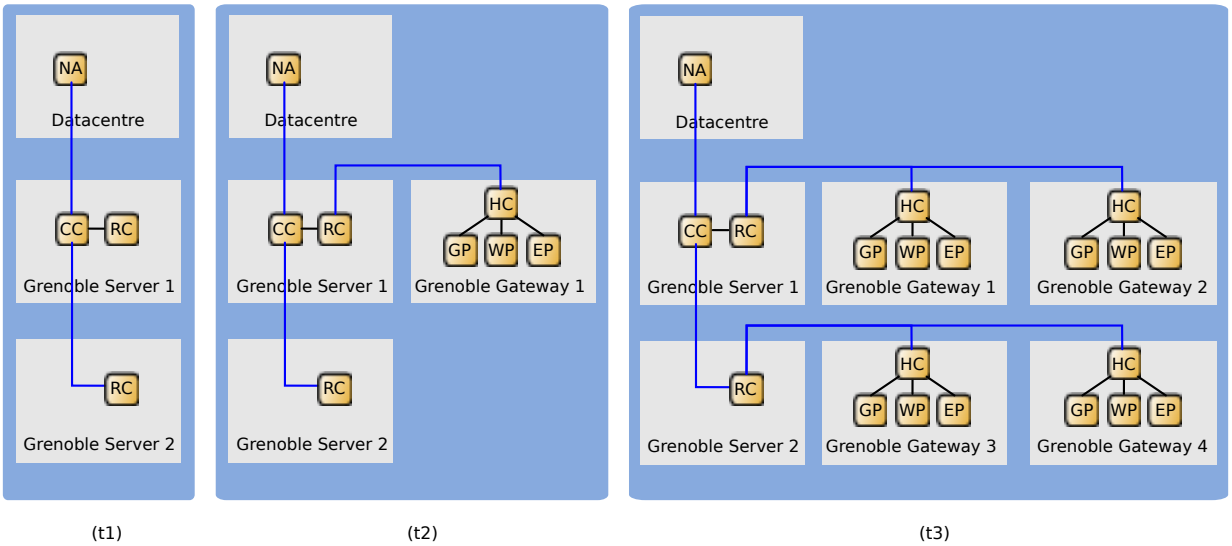


Figure 7.9: Simplified instantiation solution for the archetype of case study 1; (t1) only showing two Grenoble servers and the Datacentre; (t2) adding a Grenoble gateway; (t3) adding three Grenoble gateways.

The first depicted setting (Figure 7.9-(t1)) only includes two Server Nodes that belong to the Grenoble Scope and one Datacentre Node that belongs to the Central Scope. No Gateway Nodes are considered at this stage. In this setting, the decentralised resolution process proceeds as follows. Each Autonomic Manager (AM) of a Server Node creates an "RC" Mediator instance. They then coordinate their actions to create a single "CC" instance on one of the servers – Grenoble Server 1 in this case. Coordination is carried-out by contacting the Scope Leader of the Grenoble Scope, which they both belong to. Once the "CC" instance created, both Server AMs connect their respective "RC" instances to it.

The second depicted setting (Figure 7.9-(t2)) includes an additional Gateway Node that also belongs to the Grenoble Scope. On the Server and Datacentre Nodes the resolution process is executed as before (setting t1). On the additional Gateway Node the Autonomic Manager first creates an "HC" Mediator instance and then connects it to an "RC" instance, which it finds on one of the Servers in the Grenoble Scope. Whenever resource probes are added to the gateway – e.g. "GP", "WP" and "EP" instances in the figure – the Autonomic Manager connects them to the local "HC" instance.

Finally, the third depicted setting (Figure 7.9-(t3)) includes three more Gateway Nodes in the Grenoble Scope; resulting in a total of four Grenoble Gateways. On the Server and Datacentre Nodes the resolution process is executed as before (setting t1). On each of the Gateway Nodes the resolution process proceeds as described before (setting t2). Note here that the Autonomic Managers running on different Gateway Nodes may connect their local "HC" instances to remote "RC" instances located on different Server Nodes – e.g. the "HC" instances on Gateways 1 and 2 are connected to the "RC" instance on Server 1; and the "HC" instances on Gateways 3 and 4 are connected to the "RC" instance on Server 2. In the illustrated example, this can be due to a "hasMaxInputComponents" Goal Property defined on the "RC" component and configured with a maximum value of "2" (this case was exemplified in Figure 7.6, even if with a different value - "5"). In the absence of this constraint, different "HC" instances may still end-up connected

to “RC” instances on different Severs, since the search procedure of the various Gateway Autonomic Managers may return “RC” instances on different Sever Nodes (even if this is not the case in the prototype’s current search implementation).

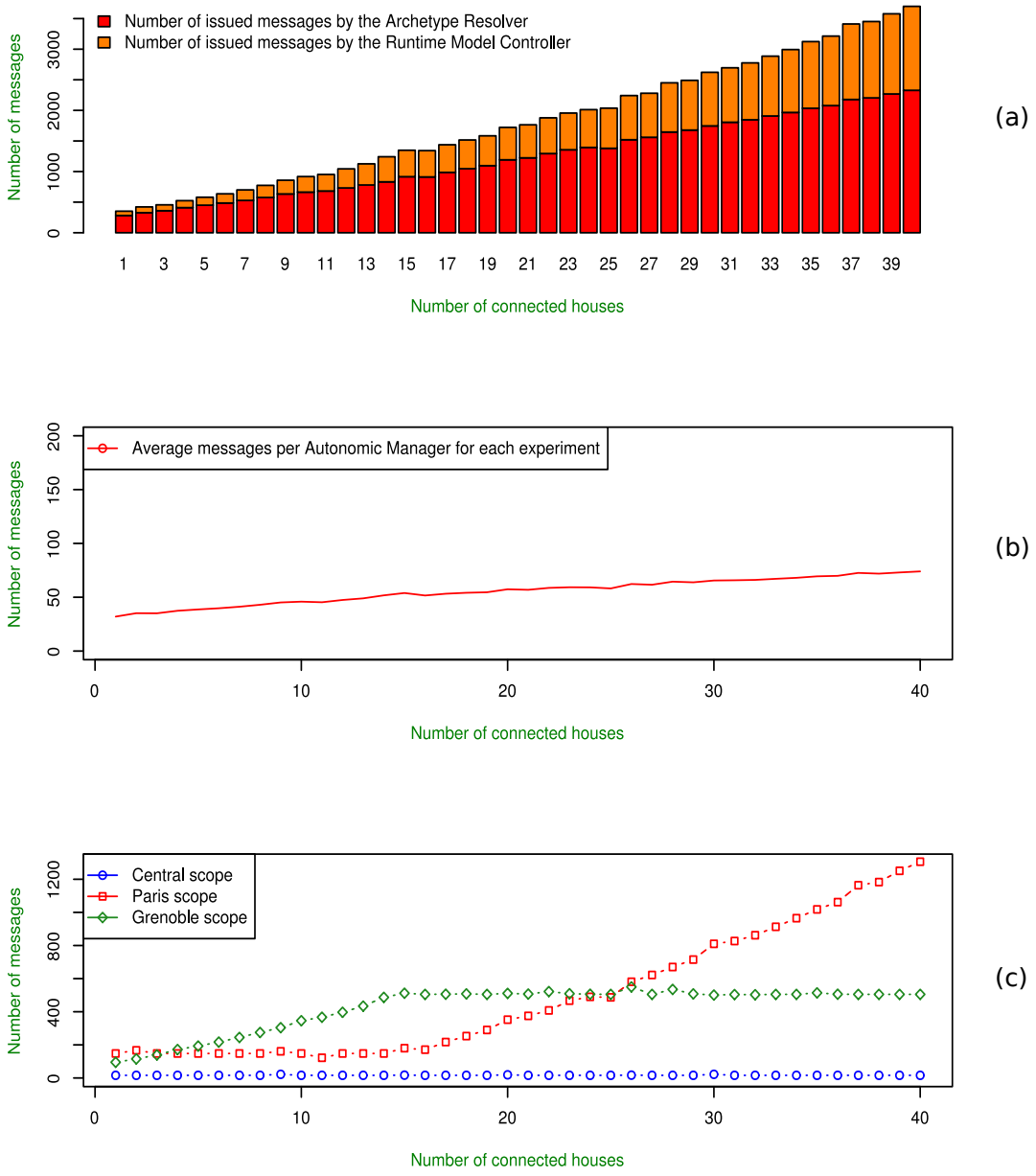


Figure 7.10: Number of control messages exchanged among Autonomic Managers (AMs) during the initial process of archetype resolution and mediation chain instantiation: (a) total number of control messages exchanged by all AMs; (b) average number of control messages per AM; (c) number of control messages exchanged by AMs located in different scopes.

Figure 7.10 depicts the results we obtained in relation to the number of control messages exchanged during the experiments. Between successive experiments the number of Gateways was increased progressively – i.e. from one to forty Home Gateways, with AM identifiers from 11 to 50, respectively. The first diagram (Figure 7.10-(a)) shows the total number of control messages exchanged by all Autonomic Managers during the initial resolution process; that is, between the moment when the Cube platform was started and the moment the distributed mediation chain was complete. We differentiate between two main types of control messages. First, we consider messages related solely to the functioning of the Archetype Resolver – i.e. direct communication between remote Archetype Resolvers. For instance, this can occur when interconnecting distributed component instances managed by the two resolvers. Second, we consider control messages related to the Runtime Model Controller - i.e. communication between remote Runtime Model Controllers. This type of communication is typically triggered by an Archetype Resolver in search for a solution that surpasses the boundaries of its local Runtime Model.

The obtained results (Figure 7.10-(a)) indicate that the total number of exchanged messages increases linearly with the total number of connected house Gateways. The additional messages that occur from one experiment to the next are generated by the Autonomic Manager of the new added house, as part of its interaction with (some of) the other Autonomic Managers in the system (e.g. scope leaders and managers of remote nodes). More precisely, adding a new house generates about 50 additional messages. These include exchanges with the Scope Master and the city Scope Leader, as well as querying the Runtime Model of other Autonomic Managers. The linear increase of communication overheads with the number of decentralised Autonomic Managers indicates the scalability of the Cube framework solution and prototype.

We'd like to emphasise here that these messages are only those needed to self-initialize the distributed mediation chain. Messages required for future adaptations are not taken into consideration at this point. Adaptation procedures will be the subject of the second case study (section 7.3). This being said, the viability and performance characteristics of adaptation operations will be very similar, or identical, to those of the initial instantiation process, since the same resolution procedures are being employed.

The second diagram (Figure 7.10-(b)) shows the average number of messages per Autonomic Manager, during each experiment. This number remains unchanged despite the increasing number of the connected houses. This result confirms the good scalability of the Cube framework with respect to the number of independent Autonomic Managers involved in the system. This is an encouraging result, since scalability was one of the main motivations behind choosing a decentralised management solution within the Cube project. In the third diagram (Figure 7.10-(c)), we analyse in more detail the control messages exchanged by Autonomic Managers located in each scope of the Cube system – Grenoble, Paris and Central scopes. As we have mentioned before, in this case study we start by connecting Grenoble Gateways, then Paris Gateways, progressively, across successive experiments.

We notice that the number of messages exchanged within the Central scope remains stable throughout the experiments. This is due to the fact that the Autonomic Manager administrating this scope is not directly affected by the addition of new Gateways; this means that it performs no additional resolution procedures when new house platforms join the system.

Conversely, within the Grenoble city scope, the number of messages issued by Autonomic Managers in this scope increases when the number of the connected houses increases. This is due to the fact that an additional Autonomic Manager is added for administrating each new house and its resolution process relies on remote communication with other Autonomic Managers (discussed above). In later experiments, when we stop adding Grenoble houses and start adding Paris houses, the number of messages in the Grenoble scope remains stable.

Within the Paris city scope, the number of messages was stable during the first experiments, when

we only had Grenoble houses connected to the system. The only few messages exchanged during these experiments were generated by the Paris Servers (5 servers). Namely, the Autonomic Manager on each Paris Server had to create a local mediation chain and connect it to a remote chain, so as to meet the related archetype goals (subsection 7.2.4.3). Hence, one "RC" component is instantiated in each Paris Server and connected to the one "CC" component instance that is instantiated on one of these Servers; in turn, the "CC" instance is connected to the "NA" component instance located in the Datacentre. In the later experiments where we start adding Paris Gateways the number of messages exchanged within the Paris Scope increases linearly.

For the very last experiment, which includes all the 40 Home Gateways - 15 in Grenoble and 25 in Paris - we have analysed the number of control messages exchanged by each Autonomic Manager (listed in table 7.3, above). Figure 7.11 shows the obtained results.

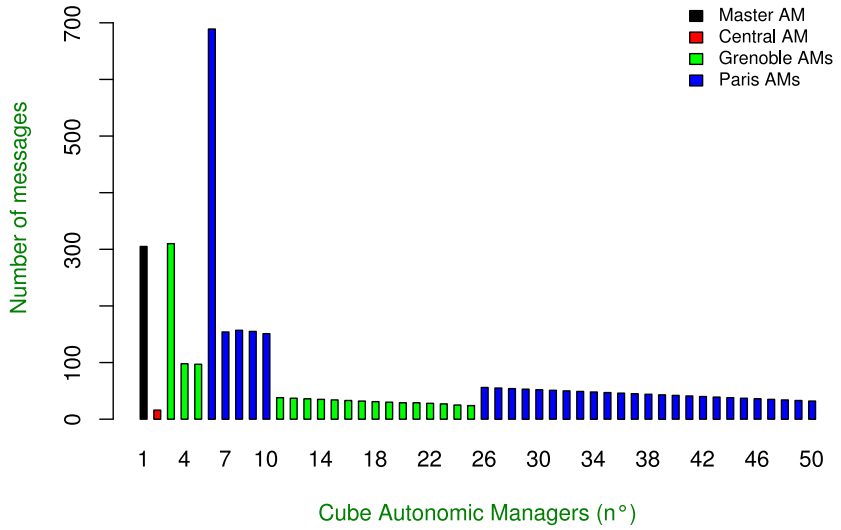


Figure 7.11: Average number of control messages for Scope formation and Archetype part resolution

The average number of exchanged control messages were approximately the same across Autonomic Managers running on home gateways, since they all performed similar tasks (AM indexes 11 to 25 for Grenoble gateways, and 26 to 50 for Paris gateways). A similar situation can be observed for Autonomic Managers managing server nodes (AM indexes 3 to 5 for Grenoble, and 6 to 10 for Paris). However, among the Autonomic Managers running on servers of the same city, one particular Autonomic Manager features a significantly higher number of control messages. These two Autonomics Managers (AM with index 3 for Grenoble and AM with index 6 for Paris) correspond to scope leaders in the respective city scopes. Therefore, in addition to resolving their corresponding archetype parts, they also respond to requests from other Autonomic Managers. Most of these requests are part of the searching process performed by Autonomic Managers for finding Nodes that fit a certain archetype description. A similar situation applies to the Master Autonomic Manager (index 1), which responds to requests for scope leaders.

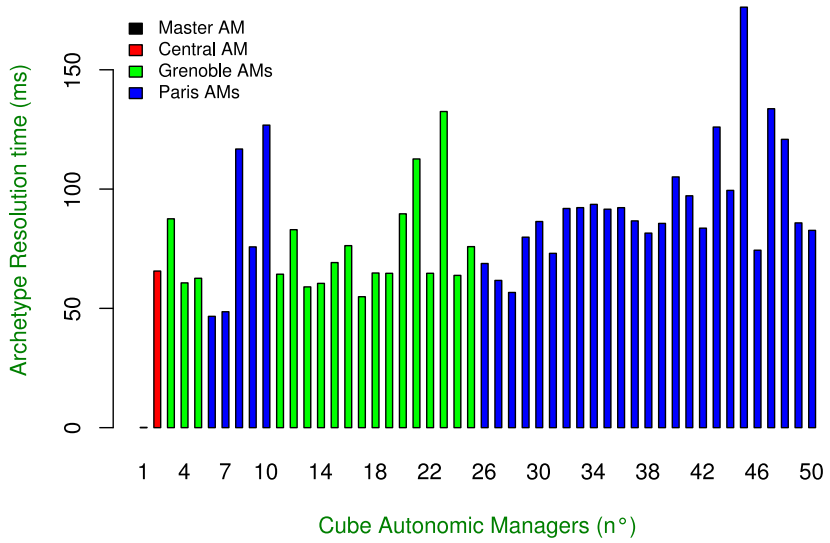


Figure 7.12: Average number of control messages for Scope formation and Archetype part resolution

Figure 7.12 shows the measured execution times that Autonomic Managers took for determining and resolving their archetype parts (or local constraint graphs). These execution times do not include the technology-specific delays for the actual creation and interconnection of Mediator instances once the corresponding Runtime Models are validated.

The first Autonomic Manager containing the Master instance features zero resolution time. This is because in the Archetype specification of this case study, there is no Goal Property associated with the Master Description Element. Also, the measured resolution time only includes actions related to the attempts of the Autonomic Manager under investigation to resolve its own archetype part; it hence excludes any time spent in collaborations with other Autonomic Managers for fulfilling their goals. The resolution times for the remaining Autonomic Managers is generally between 40 and 200 ms. For instance, the resolution time of the Autonomic Manager controlling the Datacentre (index 2) is about 65 ms.

This figure shows how the necessary resolution time (and so the self-management tasks) for resolving the entire archetype is partitioned among the different Autonomic Managers. Hence, Cube framework handles the scalability concerns raised when adding new managed platforms by adding corresponding Autonomic Managers for sharing the extra administrative load. These results show the importance of using a decentralised organisation for the self-management of large-scale mediation systems.

7.3 Case Study 2: health-care monitoring system

In this section, we describe our evaluation of the Cube framework in the context of the Medical project[3]. We begin by a short introduction of the Medical project and its objectives. We then concentrate on our contribution to this project and the use case where Cube was employed.

7.3.1 Context

Health-care at home represents an increasingly important and required service in most developed countries. Indeed, the average population of these countries has been aging progressively over the last decades and is estimated to continue this trend over the following decades. At the same time, the number of seniors residing at home, and/or, in a situation of dependence, increases due to both economic factors and personal choices.

In this societal context, it is important for governments and service providers to rely on modern information technologies for developing appropriate living environments, offering innovative healthcare services to senior individuals. Technological advancements in embedded devices, communications and digital home services can be capitalised upon to provide increasing support for this kind of health-care services.

However, introducing such services raises multiple technical challenges related to the implementation, maintenance, and evolution of such services. Basically, all services will rely on data collected from specific devices situated in the homes and transmitted over a network to healthcare-related applications (Figure 7.13). Developing, deploying and maintaining data-mediation software and infrastructure for integrating monitoring devices with service applications is a difficult task, at best.

Some of the main technical challenges are due to the heterogeneity of devices and data, the high dynamicity of devices, and the behaviour of home residents. In addition, users may want to reuse their devices when subscribing to various services, offered by different providers, which further increases heterogeneity and exacerbates the problem. Users may also change their preferences and subscribe to various providers over time, requiring the mediation logic to adapt accordingly. Finally, as healthcare services become increasingly popular, the underlying infrastructure and mediation logic will have to correspondingly scale and adapt.

These considerations indicate the need for a special-purpose mediation middleware to be introduced in-between home devices and healthcare applications. Such mediation system should be managed by a third party provider, in order to ensure a clear separation, both technical and administrative, between home residents and healthcare service providers. This third party provider would bring in the specific data-mediation expertise, hence allowing the other two parties to concentrate on their specific concerns.

For these reasons, the Medical Project - funded by the OSEO[4] and the Conseil Général de l'Isère and led by Orange Labs[5] (France) - aims to develop a solution that meets these requirements. Technically speaking, Medical focuses on the design, development and administration of an integration middleware for digital healthcare services. Since the proposed solution is rather generic, it will be applicable to a wide range of digital home services, pertaining to various domains, beyond the healthcare realm (e.g. home entertainment and security).

In addition to the actual development of such mediation middleware, its subsequent deployment and runtime management raise more significant challenges. On the user side, technological homecare services cannot be managed by residents who are not technical experts. For instance, this would involve operations

[3] http://medical.imag.fr
[4] http://www.oseo.fr
[5] http://www.orange.com/

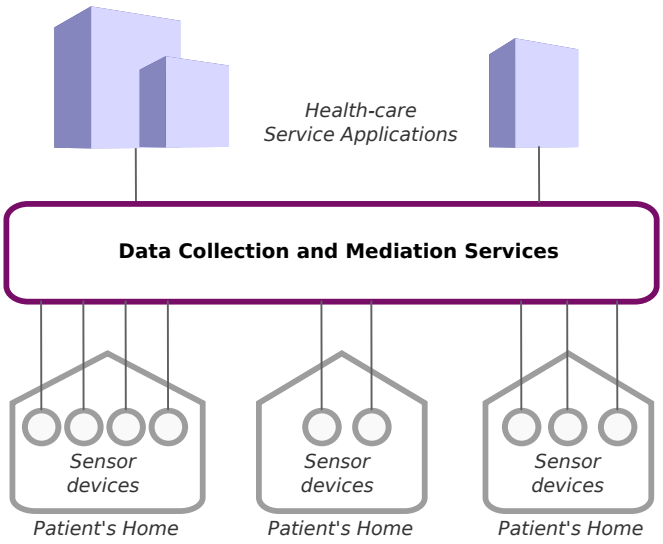


Figure 7.13: Data-mediation software for health-care service applications

such as the deployment, instantiation and configuration of specific device adapters that would inject monitored data through mediation chains customised for a targeted service. Similarly, service providers may find it difficult and costly to manually connect their services to mediation chains, potentially requiring specific adaptations. Finally, on the mediation provider side, the main challenges consist in adapting the mediation infrastructure to changing environments and evolving clients, and in repairing it when crashes occur. Therefore, Medical project aims to also endow the proposed integration middleware with autonomic management capabilities. Cube framework was adopted for addressing these issues as discussed later in this section.

Two use cases are targeted within the Medical Project:

- *Behaviour Monitoring*: aims to construct a model that represents the behaviour of patients in their homes, that would help detect anomalies, or deviations from the 'normal' behaviour, indicating potential health problems. This use case requires equipping the house with sensor devices, such as presence detectors.
- *Health-care Monitoring*: aims to facilitate access to data issued from medical sensors, like blood pressure or sugar level monitors, by doctors or family of the patient under medical surveillance. These data are collected at home and forwarded to intermediate servers for processing, persistence and comprehensive reporting.

The solution proposed by the Medical project is based on innovative technologies such as OSGi, iPOJO (both described in the previous chapter 6), OW2 Joram , and Cilia mediation framework (described in section 7.1.2). We have used Cube to provide the self-management capabilities in the context of the second use case "Healthcare Monitoring". In the next section we detail the targeted managed system, and farther on we outline the addressed problems and how Cube was used to ensure autonomic capabilities.

7.3.2 Managed System

The overall system for the Healthcare Monitoring scenario functions as follows. Physiological data are collected by sensors installed in the patient's house. These data are sent to one or several intermediate

server(s) where data is processed and persisted so that authorised professionals like medical staff or family can access and view it.

At the home level, no processing of the sensed data is performed. Rather, collected data is forwarded directly to an intermediate data-mediation provider, where it is processed (e.g. aggregated, filtered, or translated) and then sent to one or several *Personal Health Record* (PHR) provider(s), like Google Health, Microsoft Health Vault, and so on. Health professionals and the patient's family can access the processed data by subscribing and connecting to such PHR(s).

Figure 7.14 shows the mediation chain that was designed within the Medical project for the Healthcare Monitoring scenario, for processing and transporting data between home devices and PHR providers. The mediation chain is rather simple, based on a pipe-line architectural style, involving several mediators and bindings.

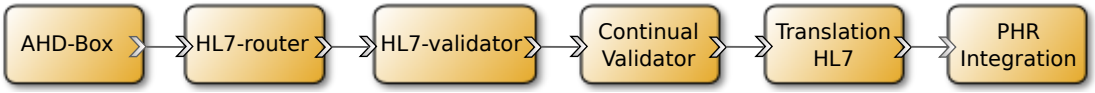


Figure 7.14: Architecture of the mediation chain of "Health-care monitoring" use case

This mediation chain executes on top of a distributed platform, or infrastructure, which is provided by the intermediate data-mediation provider and located between the patient's home and the PHR provider(s). The precise definition and functioning of each mediator in the exemplified chain is beyond the scope of our validation process. Since our focus is on the initial deployment and runtime management of the mediation chain, we are merely interested in the architecture of the mediation system and in some of its quality attributes, like availability, reliability, and so on.

The exact system settings at the home level are also beyond our self-management interests within this use case. In general, at this level, a house gateway is connected to several types of sensors, such as balance, sphygmomanometer, frequency meter, and so on. This gateway is connected to the Internet to allow sending the collected data to the intermediate mediation provider. From our perspective, the necessary sensor drivers and services are already installed in the gateway (e.g. when the gateway is activated and connected to the Internet). One important technical aspect to notice at this level is the presence of a local *Web Service* (WS) allowing the intermediate mediation provider to get the collected data via Web Service calls.

7.3.3 Problems and Requirements to address

From an architectural perspective, the mediation system presented in the previous section consists of a relatively simple mediation chain. However, in addition to application-specific functionality, administrators want to ensure the system's quality of service (QoS), preferably via administration tasks that involve little or no human interventions. We depict these objectives as follows:

1. *Initial deployment*, instantiation and configuration: automatically instantiating the mediators composing the mediation chain, on the servers provided by the distributed infrastructure. The necessary mediator types and their interconnections were specified in the previous subsection (Figure 7.14). The input data for the mediation chain is obtained from the gateways of the various participating houses. Each new integrated house should be connected to this mediation system. The output data from the mediation chain is sent to the list of PHR provider partners.
2. *Managing workload variations*: automatically performing load-balancing of incoming workloads

across several mediation servers. This implies the automatic duplication of the mediation chain across several servers so as to support overloads and to ensure that the processing time of incoming messages does not exceed a predefined threshold. Similarly, when the incoming load is decreasing, some of the servers could be stopped (and their mediation chain replicas removed) automatically. This approach can help optimise resource utilization.

3. *Integrating new PHR providers*: automatically modifying the mediation chain (and all its replicas on load-balancing servers) so that the processed data can be sent to new PHR providers, without stopping and reconfiguring the entire mediation system.

7.3.4 Solution proposed

In this subsection we detail the solution we propose for addressing the administrative problems discussed in the previous subsection. Figure 7.15 details the overall architecture of the proposed mediation system located within the "Intermediate Mediation Infrastructure". The infrastructure is divided into two main clusters. The first cluster plays the role of aggregating data from the various houses. The second cluster performs the mediation operations on the received data. As illustrated in the figure, the mediation chain is distributed across servers in both clusters, either for splitting the mediation logic or for load-balancing purposes. We use Cube to self-manage this mediation system and hence to tackle the aforementioned problems and requirements.

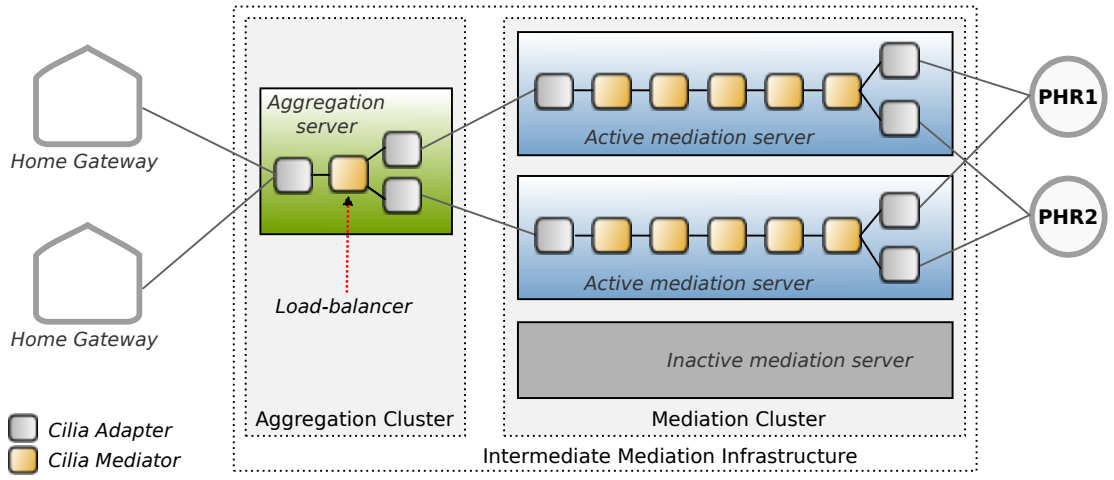


Figure 7.15: Overall mediation architecture with load-balancing between mediation servers

As previously indicated, data is collected from the participating houses using Web Service calls. We use a special-purpose Cilia Adapter for achieving this. This Adapter is inserted at the beginning of the mediation chain and configured to probe the various house gateways, in order to obtain the local sensed data. Therefore, this Adapter is deployed and executed on a server in the Aggregation Cluster (named Aggregation Server).

The collected data is then forwarded to a server in the Mediation Cluster (named Mediation Server), which contains the aforementioned mediation chain (Figure 7.14). Two further Adapters are necessary – one on the Aggregator Server (out) and one on the Mediation Server (in) - to ensure the distributed communication between these two servers. As discussed before, servers in the Mediation Cluster can be replicated (together with the mediation chains) to provide load-balancing capabilities and avoid sys-

tem overloads. A load-balancer mediator is introduced onto the Aggregation Server for distributing the collected data among Mediation Server replicas.

Data obtained at the end of the mediation chain is sent to the connected PHR providers. An additional Adapter is appended to each mediation chain (replica) for converting the processed data into a PHR-specific format. Finally, as before, this Adapter is also used to send data to each remote PHR, via specific protocols and communication technologies.

Let us now take a quick look at the way in which the load-balancing is designed in the current prototype; more details are introduced later in this section. The mediation chain performs the same tasks for each data message received. However, the processing time of each message is different (e.g. messages have different sizes). Administrators typically set in place service level agreements for guarantying that message processing induce limited latencies. To provide such guarantees, whenever a server is loaded at more than a fixed level (for instance 70%), Cube should activate another server, duplicate the mediation chain and connect it to the load-balancer (located on the aggregation server) to receive data to process. This procedure will reduce the number of messages to be processed by the overloaded server and ensures that latency remains within the acceptable interval. Conversely, when the number of input messages decreases and/or the total cluster load can be undertaken by fewer servers, Cube automatically deactivates the extra mediation servers so as to minimise the consumption of server resources.

In the following, we explain the different steps involved when using the Cube framework for managing the exemplified mediation system. We start by outlining the necessary model abstractions. Recall that model abstractions are used to represent the managed system elements at runtime (see section 5.2 of chapter 5). Then, we outline the list of extensions used, some of which were implemented specifically for this use case. Finally, we detail the proposed Archetype specification to describe the different administration goals.

7.3.4.1 Model Abstractions

As in the first case study (section 7.2), we use the model abstractions provided by the Core Extension of the Cube framework. Namely, we use the Component, Node, Scope and Master abstract managed elements to write the Archetype specification. Table 7.4 summarises the mapping of these abstractions onto concrete system elements managed in this use case. It shows how Cube framework's core abstractions are extended to represent domain-specific abstractions for modelling this specific use case. For instance, the Component concept is mapped onto Cilia Mediators and Adapters using the Cilia Extension. Additional extensions were implemented for PHR integration and Load Balancing (as discussed below).

7.3.4.2 Extensions

Table 7.5 summaries the list of the extensions used for this case study. Among them, two new extensions were implemented specifically to handle the particular requirements of this second case study. These new extensions handle the load-balancing mechanisms and the integration of new PHR providers.

Core Extension

The Core Extension provides all the model abstractions used in this use case – *Component*, *Node*, *Scope* and *Master* - as well as the majority of their related Archetype Properties (Description and Goal Properties), and their associated resolvers.

Table 7.4: Model abstractions for the second case study

Model Element	Semantic
Component	Used as an abstraction for Cilia Mediators and Adapters (like in the first case study).
Node	Used as an abstraction for OSGi platforms. Each OSGi platform contains a local mediation chain with a set of Mediators (and Adapters). It has the following two properties: • **cpu**: contains the medium value of the monitored CPU load, over a fixed interval. It is set by the Load Balancing (LB) Extension (discussed below). • **active**: if set to "true", the corresponding Node is considered to be active. This means that the managed server modelled by this Node abstraction is currently being employed in the mediation infrastructure (to host and execute mediation chains).
Scope	Used as a grouping abstraction for a set of OSGi platforms to simplify goal descriptions and to help coordinate the distributed resolution process. We use this model abstraction to regroup the OSGi platforms of the same cluster. We need to set the following property of the Scope element: • **scopeId**: the unique identifier of the Scope element. Since we only have two scopes in our case study - one for regrouping the nodes of the mediation cluster and the other for the nodes of the aggregation cluster - we do not use scope types in this use case; the identifier of each scope will suffice.
Master	Used as a global leader abstraction for coordinating the system's Scopes. In this case study, we instantiate the Master element in a dedicated Cube Autonomic Manager. It only holds the two references to the leaders of the two Scopes.

Table 7.5: Case study 2: extensions used and their configuration properties

Extension Name	Extension Namespace	Configuration Properties
Core Extension	fr.liglab.adele.cube.core	master
Cilia Extension	fr.liglab.adele.cube.cilia	jms.server
		jms.port
PHR Extension	fr.liglab.adele.cube.phr	interval
		phr-providers-file
LB Extension	fr.liglab.adele.cube.loadbalancing	interval
		max-limit

Cilia Extension

The principal role of the Cilia Extension is to synchronise the Runtime Model state in the Cube Autonomic Manager with the actual Cilia chain in the managed system (as explained in section 6.4.4 of chapter 6). Let us illustrate this here for the case where Cube decides to remove a (part of a) mediation chain. When Cube's archetype resolver removes one or several Component instances from the Runtime Model (representing Cilia mediators), the Cilia Extension consequently removes the corresponding Cilia mediators from the concrete mediation chain. To ensure consistency and avoid data losses, each Cilia mediator is only removed once all its buffered data waiting for processing have been treated and all its on-going processing operations have been completed.

PHR Extension

We proposed and implemented the PHR extension to allow integrating new PHR providers into the Mediation system. This extension reads a file located in the local file system, or on a remote server (specified in the `phr-providers-file` configuration property). This file contains the list of all PHR providers recognised by the system. The PHR Extension reads this file at predefined times (specified in the interval configuration property) in order to determine if any new provider has joined or left the system (added or removed from the PHR list, respectively). For each entry, the PHR Extension creates a new Component instance of type "PHR-integration" having a property "adapter-component" containing the name of the Cilia Adapter responsible for sending the data to that specific PHR provider. The Archetype resolver integrates this instance at the end of the mediation chain (as specified in the Archetype).

LB Extension

The load-balancing Extension (named *LB Extension*) allows activating or deactivating Nodes depending on the total load of the active mediation servers at any given time. This extension can change the "active" property value of the Node instance. We use this "active" property in the Archetype to specify what to do in the case this property changes. When the Node instance is active ("active" property is set to "true"), the mediation chain specific to the use case is instantiated and configured to receive data from the Load-Balancing Cilia mediator (located on the Aggregation Server). When the Node becomes inactive ("active" property is set to "false"), the mediation chain is stopped and removed from the corresponding Mediation Server.

The LB Extension plays the following two roles:

1. It monitors the CPU consumption of the executing servers and updates the "cpu" property of its corresponding Node instance in the Cube's Runtime Model part (Figure 7.16). In the actual LB Extension implemented, we simply provide simulated CPU values at random.
2. It calculates the global load of all the active nodes in the mediation cluster, and decides to activate or deactivate some nodes so as to remain within an acceptable load interval. This function is called at every interval (specified as a configuration property of the LB Extension).

With respect our evaluation purposes the main role of the simulated CPU values and load evaluation algorithm is to trigger Cube's capabilities for automatic replication or removal of mediation chains, hence highlighting Cube's load-balancing and system scalability features. More realistic monitoring, load estimation and server provisioning functions can be introduced in real use case scenarios.

Figure 7.16 shows a complete example of the extensions used for this use case. In this figure we do not show the Core Extension which is present in all the Autonomic Managers. A separate Autonomic

Manager is used to play the role of the Master node containing the list of scope leaders. The content of the Autonomic Managers' Runtime Model corresponds to what is specified as objectives in the Archetype (detailed next). In the Aggregation Servers, we only use the Cilia Extension (and the Core Extension) to control the local mediation chain. However, at the Mediation cluster level, we add the PHR and LB Extensions to each Autonomic Manager to ensure the load-balancing functionality and the dynamic integration of new PHR providers.

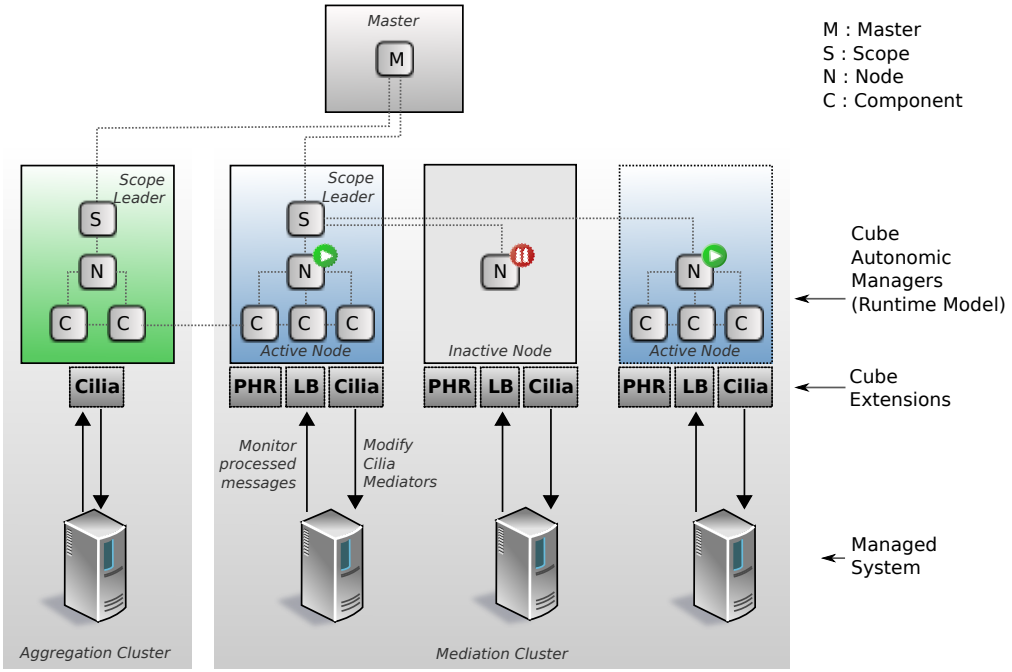


Figure 7.16: Cube Extensions for case study 2

7.3.4.3 Archetype

In this section, we detail the proposed Archetype specification for this case study. For simplicity, we divide the Archetype into several parts; each one is explained separately and is intended for a particular purpose. The entire Archetype XML file is listed in Appendix C.

The first goal to achieve is to regroup the available servers into clusters so that we can then apply common goals to each cluster and facilitate the Archetype Resolver's work. For this aim, we use the Scope element provided by the Core Extension. A Scope instance contains a list of Nodes, and is located only in the Autonomic Manager playing the scope leader role. This leader is selected dynamically, so that if the current leader fails a new one can be selected instead. Figure 7.17 shows the Archetype part that is concerned with the specification of this first goal (dynamic scope membership). The Archetype part is represented based on a graph-like format (section 5.3 of chapter 5), for clarity.

This part of the Archetype aims to assign each Server to the adequate scope, as soon as the server is integrated into the managed system. Namely, Aggregation Servers are assigned to the "AggregationCluster" scope, while Mediation Servers to the "MediationCluster" scope. In this case, we use the "inScope" Goal Property with a "Find or Create" resolution strategy. Hence, whenever a new Mediation Server is added to the system, Cube will first try to find a scope leader for the MediationCluster scope. If such scope

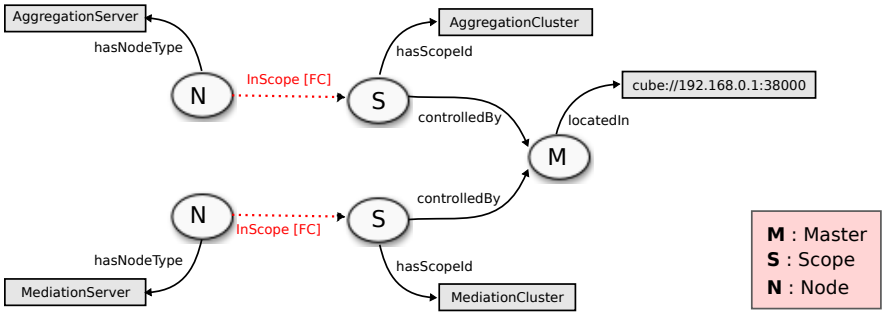


Figure 7.17: Archetype part for use case 2: dynamic scope management

leader is found, the new server is registered with the leader. This means that the scope instance will add the UUID of the new Node instance to the list of nodes members of this scope. Otherwise, Cube creates a new scope instance which will be the scope's leader. This instance is placed in the Autonomic Manager of the newly integrated Server. The two scopes are controlled by a Master instance located (statically) in Cube's Autonomic Manager identified by a predefined URI (`cube://192.168.0.1:38000`).

Figure 7.18 shows how Component types are specified to form the first part of the mediation chain, which is to be located on the Aggregation Server. In this scenario, we want Cube to create this mediation chain automatically, as soon as the Aggregation Server is activated. Therefore, we use the "hasComponent" Goal Property, and the associated "Create" strategy. This binary Goal Property takes as parameters the description of the Aggregation Server and the description of the first Component ("WSProbe") of the mediation chain. It indicates that a "WSProbe" Component must be created on each Aggregation Server. Figure 7.18 illustrates this Goal Property via an arc connecting the Node (N) - that "hasNodeType" "AggregationServer" – to the Component (C) – that "hasComponentType" "WSProbe". This Component's description also includes the location of the Web Service Sources file, which contains the addresses of the Web Service Endpoints representing the connected houses ("hasWSSources" Description Property).

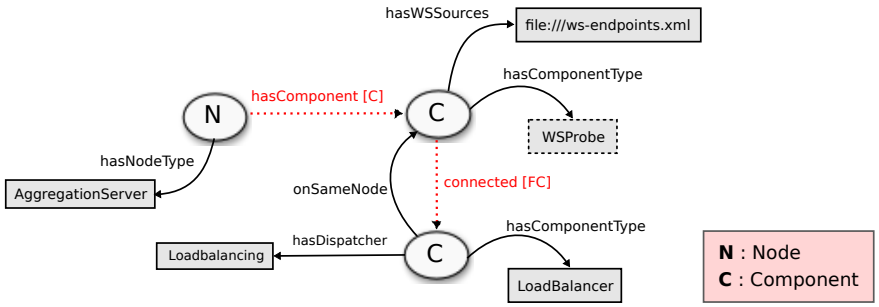


Figure 7.18: Archetype part for use case 2: self-creating components on the Aggregation Server

When the "WSProbe" Component is created, we should connect it to another Component that can dispatch data messages to the mediation infrastructure. We use the "connected" Goal Property for this aim. The object element of this goal is a description of the required Component, which is of type "LoadBalancer". Note that we also added the "hasDispatcher" Description Property to it, with the associated value "LoadBalancing", to make sure it provides the necessary load-balancing functions. The effect of this description is the addition and setting of a particular Cilia property in the newly created Component

instance; this property is called “dispatcher” and will be assigned a “loadbalancing” value. This model property is then used by the Cilia Extension when instantiating the corresponding Mediator, in order to set a Load-balancing Cilia Dispatcher for this Mediator.

Once the LoadBalancer Mediator is added, the aggregated data is sent to a different destination (Mediation Server) each time. In reality, an additional Component of a specific Adapter type must also be used to ensure remote communication; this will be created automatically by Cilia extension when detecting remote references. Finally, we also added the “onSameNode” Description Property between the two Component types, in order to force the “LoadBalancer” Component instance to be created on the same server as the “WSProbe” Component instance.

At the Mediation Server level, we set up the remaining mediation chain as described before (Figure 7.8bis). Some additional considerations must be taken into account at this point. First, the foremost mediator in this chain should connect to the Aggregation Server, in order to receive the aggregated data collected from houses. Second, once this first mediator is created, the remaining mediators in the chain should be created automatically, so as to correspond to the targeted architecture. Last, we should be able to add new PHR providers without stopping the mediation infrastructure. Figure 7.19 shows the corresponding Archetype part for ensuring the two first objectives, while Figure 7.20 shows the corresponding Archetype part for ensuring the third objective.

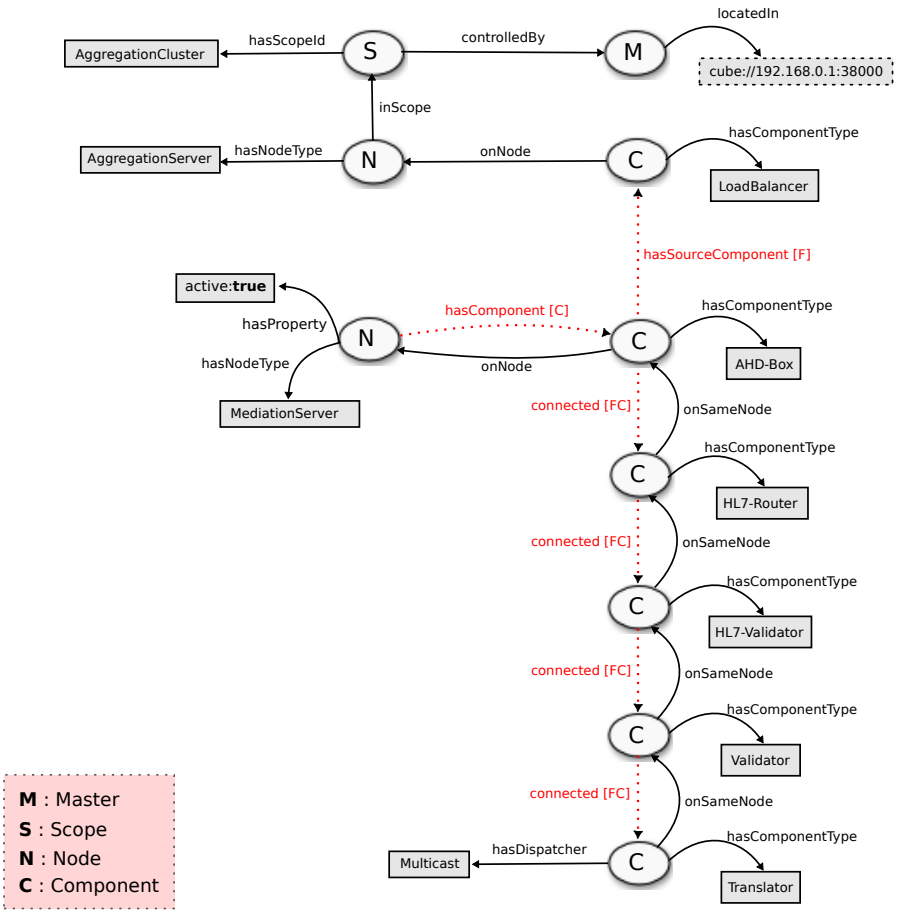


Figure 7.19: Archetype part for use case 2: self-creating components on each Mediation Server

Whenever a Node instance representing an active Mediation Server is added to the Runtime Model, Cube automatically creates a new "AHD-Box" Component instance (as seen before for the Aggregation Server, based on the "hasComponent" Goal Property). This new Component instance should now be added to the list of destinations of the "LoadBalancer" Component instance, which is located on the Aggregation Server (discussed above). We use the "hasSourceComponent" Goal Property (with an associated Find [F] strategy) to specify this objective. To reach this objective, Cube looks for a "LoadBalancer" Component instance that matches the Description Properties provided in the Archetype for the "LoadBalancer" element. These properties indicate that the "LoadBalancer" Component is located on a Node that has "AggregationServer" as type, and which is located within a Scope identified by "AggregationCluster". The Master address from which Scope instances can be retrieved using their unique identifiers is also specified in the Archetype (using the "controlledBy" Description Property for the Scope).

The same "AHD-Box" Component instance should also be connected to the next mediator - called "HL7-Router". This mediator Component is constrained to be located on the same Node as the "AHD-Box" mediator Component. The same location constraint is defined for all the other mediation Components in the chain. Finally, the last mediator Component, "Translator", is created with a "multicast" dispatcher to allow sending a copy of the processed data to all the available PHR integration components (as discussed next).

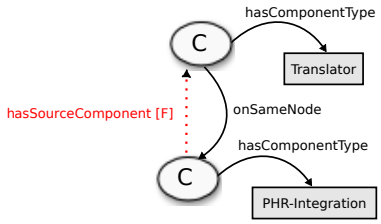


Figure 7.20: Archetype part for use case 2: self-integrating new PHR providers

To install a new PHR provider, the PHR Extension is configured to pull periodically a specific file containing the list of providers (phr-providers-file configuration property). On each Mediation Server, the PHR Extension creates a "PHR Integration" Component instance in the local Runtime Model, with properties related to the manner in which the data is addressed to the PHR provider (e.g. the PHR provider's Web Service endpoint). When created, the "PHR Integration" Component instance is automatically connected to the last mediator of the mediation chain - "Translator". We de describe this objective using the Archetype part depicted in Figure 7.20.

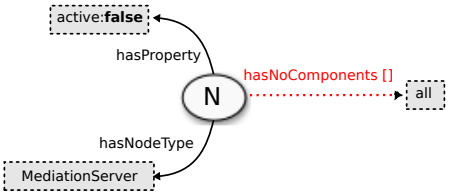


Figure 7.21: Archetype part for use case 2: self-removing mediation chains from inactive servers

Finally, to handle the case in which a Mediation Server is deactivated by the LB Extension (load-balancing), we add the following archetype specification (Figure 7.21). If a Mediation Server's Node

instance becomes inactive ("active" property is set to "false"), then Cube removes all the Component instances from the local Runtime Model. This will imply removing the corresponding Cilia mediators from the local OSGi platform. This behaviour is obtained by using the "hasNoComponents" Goal Property on all Nodes of type "MediatorServer" having an active property set to "false".

7.3.5 Scenarios and results

To evaluate the viability of the proposed solution we executed several OSGi platforms in parallel, on one machine, to simulate the different Aggregation and Mediation servers. We have used the following server configurations (Table 7.6):

Table 7.6: Testing configuration for case study 2

AM n°	Symbolic Name	Role
1	master	Autonomic Manager used to host the Master instance. It has the following url: cube://localhost:38000
2	aggsrv	Used to manage the single Aggregation Server.
3	medsrv1	Used to manage the first Mediation Server.
4	medsrv2	Used to manage the second Mediation Server.
5	medsrv3	Used to manage the third Mediation Server.
6	medsrv4	Used to manage the fourth Mediation Server.

The main purpose of this evaluation was to determine Cube's capability to: scale the mediation application up and down depending on fluctuating workloads (load-balancing function); extend the mediation chain to support new PHRs (self-adaptation function); and, repair the mediation chain in case of server failures (self-repair function).

To initiate a testing session, we always start by launching the "master" Autonomic Manager. Then, all the other AMs can be started in any order.

When the "aggsrv" server starts, it is the only server of the Aggregation Cluster. Hence, the Autonomic Manager controlling it is designated to be a scope leader - it hosts the Scope instance having "AggregationCluster" as identifier.

When the Mediation Servers start, the first one ("medsrv1") is instantiated with the "active" property set to "true", while the three other ones ("medsrv2", "medsrv3" and "medsrv4") with the "active" property set to "false". This implies that only the "medsrv1" Mediation Server is active at start-up.

Load-Balancing

Figure 7.22 (a, b, c) shows different states of the four servers in the Mediation Cluster at different times. For each state, the grey rectangles represent an inactive mediation server, while the white rectangles represent an active mediation server. In the first state (a), a single mediation server is activated ("medsrv1"). Its CPU load (17%) does not exceed the predefined threshold (70%). In the second state (b), the server's CPU load has exceeded this limit, triggering Cube to activate another mediation server ("medsrv2"). Since the activation of this new mediation server did not suffice for bringing the average server load beneath the fixed threshold, Cube has activated a third mediation server ("medsrv3"). In this new state, the medium CPU load on each of the three active servers does not exceed the identified threshold (70%). In the last case (c), all the mediation servers were activated to support the load overhead.

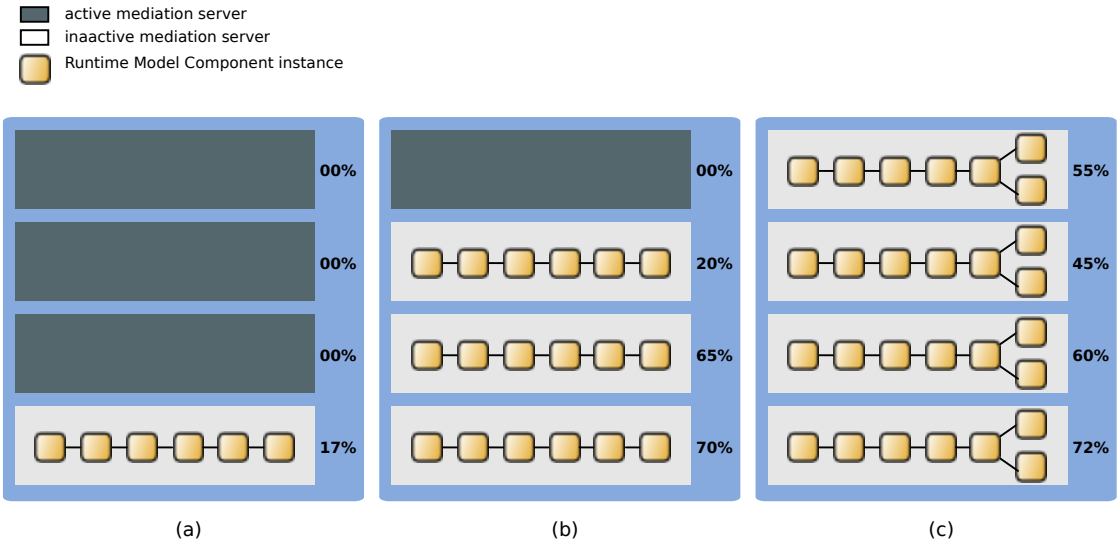


Figure 7.22: Case study 2: load-balancing based on the average CPU load

If the total load still exceeded the fixed limit, and if there were no extra mediation servers to activate, Cube would show a corresponding administrative message in the console, indicating the problem and pointing system administrators to remedy the situation. In an actual production system, Cube may be programmed to send an email to the administrators directly, to inform them about the current situation.

Self-Adaptation

In the case presented in Figure 7.22-(c) a new PHR provider was integrated with the mediation system. This required Cube to adapt the mediation chain by adding a specific integration mediator for the new PHR. This change had to be performed for all the chain replicas executing on all the active mediation servers. As described before (in the 'Archetype' part of subsection 7.3.4), a PHR Extension is associated to each Autonomic Manager of a Mediation Server. This PHR Extension detects the addition of a new PHR provider and creates a corresponding Component instance for enabling its integration (see the "PHR-Integration" Component Type defined in the Archetype part in Figure 7.20).

When this component instance is added Cube analyses the Archetype and finds that any instance of this component type must be connected to a source component (of Type "Translator") for its data provisioning – see the "hasSourceComponent" Goal Property in Figure 7.20. Hence, Cube must modify the local mediation chain so that the processed data is also forwarded to the new "PHR-Integration" component and so to the new PHR provider. This runtime adaptation is applied locally, with no communication between Autonomic Managers, since the newly created component and the rest of the mediation chain reside on the same Node.

Self-repair

Let us now illustrate the self-healing capability of the mediation cluster when managed by the Cube framework (Figure 7.23). Continuing the previous scenario, we now simulate that one Mediation Server goes down. Here, we differentiate between two configurations: (1) the Cube Autonomic Manager controlling

the broken Server is located on a different machine than the one the Server was executing on; and (2) the Cube Autonomic Manager and the Server it controls execute on the same machine.

In the first configuration (AM and Server on different machines), the Cube Autonomic Manager detects automatically that the server is broken. It then notifies the neighbouring AMs (those sharing references with the considered AM) about the incident. Notified AMs will remove from their Runtime Models all remote references to modelled instances located in the Runtime Model of the concerned AM.

In the second configuration (AM and Server on the same machine), the Cube Autonomic Manager goes down together with the broken server, without getting a chance to inform the other AMs about the incident. In this case, it is up to the neighbouring AMs (having references to instances located on the broken AM) to detect the departure of the remote server. As described in subsection 5.4.4 of chapter 5, this is precisely the role of the Autonomic Manager's "LifeController", which probes all neighbouring AMs periodically, to check their aliveness.

In both configurations above, if the AM of the broken Server is also a Scope Leader (Figure 7.23-(a)) then the other AM members of this Scope will also detect the departure of their current Scope Leader; also via their "LifeController". In this case, in the Runtime Models of these AMs, all the Node instances belonging to that Scope will have a missing reference to their required Scope (Leader) instance. Hence, the Archetype Resolver of each AM will try to find a new Scope instance (Scope Leader), by interrogating the Scope Master. Only one of the AMs in the Scope will create the scope instance, and become a Scope Leader; this will be the first AM to interrogate the Scope Master. The other AMs in the Scope will make a reference to the new scope instance.

At this point, a new Scope Leader is elected and the cluster's load-balancer is reconfigured to no longer send data to the broken machine (Figure 7.23-(b)). Now, if the load on the remaining mediation servers is determined to exceed the predefined threshold, the Load Balancing Extension will activate additional Mediation Servers (if available, as before) to absorb the additional load (Figure 7.23-(c)).

The cluster is now self-repaired, being able to provide the same mediation service as before the Server crash.

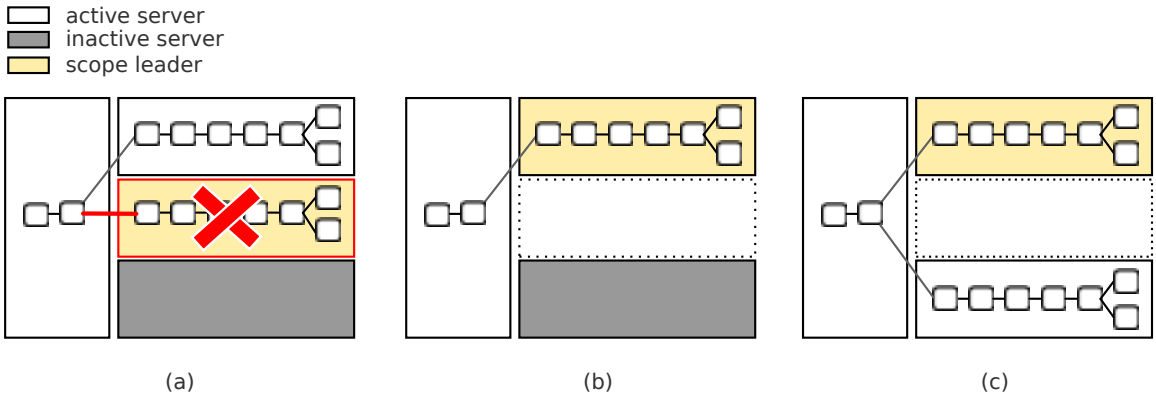


Figure 7.23: Case study 2: self-healing of the mediation cluster

7.4 Evaluation

In this section, we evaluate the proposed Cube framework based on the experimental results obtained from the two use cases. Our evaluation analysis focuses on qualitative metrics related to the functioning of the Cube system - represented by the set of Autonomic Managers as well as the managed system. That is, we evaluate the quality attributes of the developed framework, as well as the impact of using Cube framework on the management of targeted systems. Different qualitative attributes were outlined and used as comparison metrics between existing Autonomic Computing frameworks in chapter 2 (section 2.5.1). In the following, we evaluate our proposed framework against these metrics, based on the results obtained and illustrated in the developed case cases.

Dynamic Adaptability

Dynamic Adaptability refers to the capability of a system to change during runtime. Such changes can consist in adding, removing, or updating components and services, in order to adapt to modifications in the user's requirements or in the runtime conditions.

The Cube Autonomic Manager (AM) was designed to be adaptable during runtime. The different extension points of the Cube AM follow a service-oriented architecture and are implemented on top of the OSGi platform - the dynamic execution platform for Java. Extensions can be added and removed dynamically to and from the AM without stopping it. The Cube framework provides a dedicated API for handling the dynamic installation, removal or reconfiguration of AM extensions.

The same level of flexibility is also provided to handle the different AMs themselves. This means that Cube AMs can be created, stopped, started, and removed dynamically without impacting the global functioning of the system. If a removed AM has a certain self-management responsibility, Cube finds another AM with equivalent capabilities automatically, in order to solve the problems caused by the departure of the initial AM. Nonetheless, while this is an important self-management capability, we have not yet explored it in the studied use cases. Cube Autonomic Managers and their Extensions are declared and created at start-up. Some AMs were created dynamically for testing and simulation purposes only. We discuss future works using this facility in the conclusion chapter .

In the presented use cases, we have focused on highlighting the dynamic adaptability of the managed system, which is at the core of any self-management capability. In Cube, this capability relies mostly on the dynamic adaptability of the Runtime Model. The content (Runtime Model element instances) is dynamically updated by technology-specific monitors, so as to represent the current state of the managed sub-system. It is also modified dynamically by the Archetype Resolver to validate the monitored state, or, in case the state cannot be validated, to propose new solutions. Technology-specific executors reflect the Runtime Model changes into the managed system. This renders the managed system itself dynamically adaptable.

Scalability

Cube features a decentralized architecture consisting of a set of Autonomic Managers, which share and strive to reach collectively the same goal specification. This decentralized self-organised approach is a key driver of the scalability of the proposed autonomic solution.

However, actually proving that a system is really scalable can be quite challenging. The first obstacle is the lack of a common conceptual understanding of scalability [DRW06]. Scalability can refer to different aspects depending on each concerned system. It can refer for instance to the ability of a network to

support the addition of nodes; or the ability of a server system to always respond satisfactorily despite an increasing number of requests.

In the context of our thesis, we regard scalability from a management point of view. Indeed, a new class of software systems are coming about, which consist of large numbers of software and hardware entities, distributed geographically and across numerous platforms, and heterogeneous. In addition, the size and deployment scope of such systems seem to follow a general trend of continuous growth and extension.. Managing and administrating such systems become a difficult and costly task.

Hence, autonomic systems should be scalable in the sense of supporting such system growth tendencies in size and complexity. To address the scalability challenge, the main features we consider in the Cube framework are loose coupling and decentralisation of Autonomic Managers. Loose coupling is an important aspect for the autonomic system's flexibility, allowing it to be changed without major difficulties. Decentralisation is essential for dealing with large numbers of managed system resources and for reacting to changes that occur at high frequencies.

We consider the loose coupling in several parts of the Cube framework: Runtime Model, Communication service, and Administration service.

At the Runtime Model level, loose coupling was considered when designing the Modelling framework (chapter 5, section 5.2.4). In fact, the modelled element instances of the Runtime Model are related between them using their unique identifiers (UUID). There are no direct Java references between these objects. To illustrate the advantage of this approach, let us consider that one Runtime Model element instance has a reference (UUID) to another local element instance. If the referenced element instance moves to another Cube Autonomic Manager there is no need to update the value of the reference field in the refereeing element instance. Hence, irrespectively of whether referenced instances are local or remote, all references are based solely on UUID identifiers; no Java references and no Java interfaces are used between instances.

The second loose coupling aspect is related to the Communication service. Indeed, Cube uses a message-driven, non-blocking (asynchronous) communication model between the different Autonomic Managers (see section 5.4.6 of chapter 5). We have specified a common message format and a unique communication interface allowing the adoption of different communication implementations and protocols respecting this message format. This enables Cube administrators to choose the adequate communication module case-by-case, depending on the targeted managed system, its deployment architecture and network constraints. In this manner, the AMs' communication-related aspects are separated from the self-management aspects.

In all cases, there are no open connection sessions between Autonomic Managers. However, each Autonomic Manager should monitor neighbouring AMs (managing adjacent parts of the overall solution). This is done by exchanging messages via specifically defined protocol.

Cube's decentralised architecture based on loosely-coupled Autonomic Managers allows scaling the Cube system horizontally, by seamlessly duplicating the number of Autonomic Managers, so as to control increasing numbers of managed distributed sub-systems; this capability was demonstrated in the first use case (subsection xx). It also improves the overall reliability and robustness of the Cube system, since any one AM can be replaced dynamically by another AM.

There is no central controller for coordinating the set of Autonomic Managers. Instead, each AM controls a part of the managed system and collaborates with its AM neighbours (those participating in self-managing adjacent system parts).This also implies that there is no central knowledge, hence removing the need for a large central data store. Instead, each Autonomic Manager maintains its local knowledge, which it interconnects with the local knowledge of neighbouring AMs that it collaborates with. Indeed, constructing and maintaining a central model of the entire managed system would be a highly complex

and costly task, due to the large scale of the deployment platform, and to the recurrent changes of the managed sub-systems and their execution environments.

A possible downside of decentralised approaches has been known to lie in the communication overheads induced by its coordination procedures. To circumvent this potential inconvenience in the Cube framework, we have adopted a hierarchical solution for managing situations where remote coordination is needed (among AMs that are not neighbours). Namely, we have introduced federation-like management at the scope level (via scope leaders and master) to help achieve Goal Properties involving non-neighbouring AMs. Their use was illustrated in the presented use cases (subsections 7.2 and 7.3). This hybrid approach capitalises on decentralised control for scalability and robustness (discussed next) purposes, while also taking advantage of the efficiency of hierarchical control to avoid coordination overheads among large numbers of distributed AMs.

Alternatively, a completely decentralised solution can also be adopted to implement the overall Cube approach. The particular design solution for each Cube framework will highly depend on the characteristics and requirements of each managed system and application domain.

Robustness

The robustness quality metric is also related to the autonomic property: it relies, among others, on the system's self-healing capability, which re-establishes the managed systems in case of failures.

Cube framework supports the self-healing of the targeted managed systems, by continuously monitoring the system's state, comparing it to the archetype and performing necessary changes to maintain its validity. In addition, Cube framework provides self-healing support for its Autonomic Managers, by incorporating a dedicated mechanism to detect the aliveness of collaborating Autonomic Managers (see the functioning of the Life Controller module, section 5.4.4 of chapter 5). Namely, if one Autonomic Manager fails, its neighbouring AMs discover its absence and try to find alternative solutions with other AMs. This renders Cube system resilient to failures and hence improves its robustness and efficiency.

The decentralisation of the autonomic management process avoids the single point of failure problems. This aspect was demonstrated in our second use case (subsection 7.3). We have tested the case in which a failure occurs in an Autonomic Manager that plays the role of scope leader. When this AM breaks down, its neighbouring Autonomic Managers become aware of that (via their Life Controller module). They subsequently try to find a new scope leader, by creating a new scope instance in one of the Autonomic Managers belonging to the same scope.

Interoperability

Interoperability is defined as "*the ability of two or more systems or components to exchange information and to use the information that has been exchanged*" [EE90]. In the context of Cube, interoperability is considered as the ability of specifying generic goals involving different heterogeneous sub-systems. This is achieved via Cube's Modelling framework and Archetype Specification. Each technology-specific monitor will represent the monitored sub-system state in the Cube's Runtime Model. This enables all modelled element instances to be represented in a coherent manner and to have the same class-base. Developers can implement Goal Properties and corresponding resolvers independently from these different technologies, by only reading and modifying the 'standardised' Runtime Model element instances. Cube's Runtime Model is considered to be a common means of representing and sharing knowledge about various heterogeneous sub-systems. Knowledge represented in this way can then be used by goal resolvers to find coherent solutions, in a technology-independent way. These are then translated back into heterogeneous solutions via technology-specific executors. We have demonstrated this capability in the second use case

when we have specified goals that involved two different sub-systems: the mediation chain (application level) and the mediation servers (hardware level).

Furthermore, the local knowledge that is contained in the Runtime Model of one Autonomic Manager can be exploited by other Autonomic Managers looking for solutions to their local problems. We have demonstrated this distributed interoperability capability in the use cases when we imposed that one component in one server had to be connected to another component located in another server. This type of constraint demands that the AM hosting the initial component requests other AMs to search their local Runtime Models for the targeted component; if this is found the two AMs connect their partial solutions and become neighbours.

In conclusion, Cube's generic Runtime Model format and Communication protocol enable AMs that manage heterogeneous system parts to interoperate for achieving a global solution for the managed system.

Reusability

Reusability refers to the suitability of components and subsystems across various applications and/or use cases. Reusability helps prevent the duplication of components and hence can reduce the implementation time of new applications. Cube was designed with reusability as a first-class requirement. This means that applying Cube for managing specific use cases should take relatively little development time, as the different autonomic components that are often highly difficult and costly to build and integrate can be simply reused from previous scenarios. These include support for knowledge representation and management (the Runtime Model), AM coordination (via Communication protocol and coordination mechanisms), AM self-repair (via the Life Controller), and integration logic (such as the generic Resolver and support for specific Extension plug-ins). Hence, reusing autonomic components across different autonomic systems is highly demanded.

We have seen in chapter 6 the extension mechanisms provided by the Cube framework. These represent different extension points where developers can provide new implementations for specific cases. Such new implementations can then be reused across subsequent projects. We have proven this capability via the two developed use cases detailed in this chapter.

In the first use case, a single new extension (Cilia Extension) was implemented to allow the Cube framework to self-manage mediation chains implemented using Cilia technology. Specific Cilia concepts were directly mapped within the existing abstraction models (i.e. within the Core Extension). Thus, there was no need to provide any new model abstractions.

Concerning the code developed for the second use case, the only new code that we had to add was for the two new Extensions - PHR (Personal Health Record) and LB (Load Balancing). Together, the two extensions do not exceed 100 lines of Java code. In addition to this, we had to write the application-specific Archetype Specification (less than 50 lines of XML code) and the set of configuration files (less than 50 lines of XML and text code). This is considered to be a very small quantity of extra-code.

Generally, the time necessary to adapt Cube for usage in these specific use cases was quite small indeed. This was made possible by the straightforward reuse of already available Cube extensions and tools. Particularly, we used the Core Extension's abstract models and their related Archetype properties for our special cases as it totally fitted our requirements for managing the application's architecture. In the case of the first case study, we can add to the necessary extra-code the development of the Cilia Extension. Then we could use the same Cilia Extension for the second use case as it was concerned with the same underlying mediation technology: Cilia.

Generality

The last quality metric that we evaluate is the generality of our approach.. Indeed, while the data-mediation systems exemplified in the two use cases were both based on Cilia technology, we believe strongly that the proposed Cube framework can apply relatively easily to data-mediation systems based on different technologies. As we described in chapter 6 (section 6.4 "extensions") and as we demonstrated in the two use cases, Cube is designed to manage high system heterogeneity. Indeed, managed software systems can be composed of different, heterogeneous sub-systems. Managing such systems requires experts from different domains. We have proposed Cube and its Archetype specification to allow these various experts to integrate their efforts by writing a common archetype specification for managing the entire system; and introducing technology-specific extensions for translating between the generic solutions proposed by Cube and the actual system resources of various types.

Furthermore, the modelling framework introduced in Cube allows representing and then self-managing different levels of the software system, from low level hardware devices to high level software applications. This allows mastering administration objectives across all levels.

Most of the concepts, mechanisms and designs proposed for the Cube framework are pertinent, most likely, in application domains beyond data mediation. Nonetheless, since the use cases studied during this thesis focused solely on the data-mediation realm we limited our claims to this application area. Investigations on the applicability of Cube framework in other application domains is left for future research.

7.5 Summary

In this chapter, we have evaluated our proposed Cube framework in two different case studies as part of national research projects with industrial partners. In the first use case - monitoring the consumption of home resources - we conducted a quantitative evaluation by measuring the self-management operations undertaken by the Cube framework to ensure the self-creation of a distributed data mediation application. We showed that such complex management operation can necessitate only a limited time to be carried-out, when compared to manual or ad-hoc creation and configuration of Cilia mediation chains using tools provided by Cilia. In the second use case - health-care monitoring system - we conducted a qualitative evaluation of our framework. We addressed self-management operations involving different aspects. We concluded this chapter via a discussion and evaluation of the quality attributes of our framework.

Chapter 8

Conclusion

Contents

8.1 Summary **187**
8.2 Summary of thesis contributions **189**
8.3 Limitations and perspectives **190**
8.3.1 Archetype Specification 190
8.3.2 Runtime Autonomic Management 191
8.3.3 Towards a reusable methodology and integrated development environment for engineering autonomic computing systems 195

In this final chapter, we summarise the problems addressed in this thesis, we enumerate our contributions and discuss the remaining issues and limitations of our solution. We then conclude and outline some perspectives and future work that would improve the capabilities of the proposed Cube framework.

8.1 Summary

Nowadays, software systems are becoming increasingly heterogeneous and distributed, involving several interconnected execution platforms, like data-centres, personal computers, smart-phones, home-boxes and sensors. Hence, system computation is dispersed across a variety of devices, featuring heterogeneous technologies and data, and is based on an increasing number of integrated subsystems, each taking over specific tasks. While many system parts consist of legacy applications, other parts are more open and adaptive - they can operate in highly-dynamic conditions experiencing frequent changes in their internal constituents and execution environment. Managing this type of software systems is becoming increasingly difficult and costly. It involves complex management tasks, which are currently performed by either human administrators - bringing in the risk of configuration errors and low reactivity - or ad-hoc management applications - which are in turn difficult to develop and maintain. Both solutions increase system maintenance costs significantly.

Autonomic computing has been proposed as an approach to tackle the increasing complexity of managing modern-day software systems. It aims to provide self-management capabilities to software systems, including self-configuration, self-healing, self-optimisation and self-protection. The autonomic computing initiative was detailed in chapter 2 of this thesis. In particular, we have seen that the logical reference architecture proposed by the autonomic computing community has profoundly impacted this research domain and the related research efforts. This reference architecture relies on four main data-processing

stages – monitoring, analysis, planning and execution – interconnected into a control feedback loop between system sensors and actuators. All processing functions share a common knowledge repository. Different techniques and tools have been investigated and proposed in the state-of-the-art to help the development of the four management functions and of knowledge base. Yet, the reference architecture of autonomic computing remains at a high level of abstraction and hence requires more concrete designs and implementations to be developed case-by-case.

Making complex computing systems self-managed involves more internal complexity. It is likely that autonomic elements will have complex lifecycles, continuously carrying out multiple threads of activity, and constantly sensing and responding to their execution environment. It is clear that engineering such autonomic computing systems is an essential issue. Hence, to obtain effective autonomic systems, along with the corresponding self-* properties, the autonomic management logic should also feature, like any other modern software, certain quality properties such as adaptability, dynamicity, scalability, robustness, interoperability, generality and reusability.

In chapter 2 we have studied a significant list of autonomic frameworks proposed in the state-of-the-art, from a software engineering point of view, and we have established a way to compare them. The most important limitations in existing works include: lack of scalability, and the associated performance degradation; lack of support for adapting the autonomic management logic; lack of extensibility for supporting the heterogeneity of managed elements in various domains; and, lack of software engineering techniques enabling modularity and reuse.

To remedy this situation, we have studied in more depth several research approaches related to autonomic computing (chapter 3) including control theory, agent-based systems, nature-inspired applications, and architecture-based adaptation approaches. Several techniques used in our thesis proposal are inspired from these related domains. Overall, the generic approach – the Cube project - that this thesis relies on adopts and merges together several concepts from relevant domains, in order to form a generic solution for self-managing modern software systems. These include the use of abstract architectural models to specify system constraints and the decentralised context-aware reification of such models into concrete computing systems, which can hence adapt to their environments while meeting predefined goals. This thesis takes this approach forward and proposes a reusable framework – the Cube framework - for developing autonomic systems based on these ideas. The thesis also provides a series of framework extensions for the data-mediation application domain.

In chapter 4, we have briefly presented our contribution to the self-management of dynamic, heterogeneous and large-scale software systems: the Cube framework. The proposed solution uses architectural models to allow the right level of abstraction in knowledge representation and reasoning, and provides common grounds for the integration of heterogeneous technologies and their associated evolutions and extensions. It also relies on decentralised organisation of autonomic management logic to ensure better system scalability and robustness. Namely, each Autonomic Manager only maintains a partial architectural model during runtime. Managing different system parts individually and only maintaining partial views of the running system within each Autonomic Manager contributes to the overall scalability and survivability of the framework, and consequently of the entire autonomic system. To facilitate cooperation among decentralised processes, Autonomic Managers contributing to a global system property are dynamically set to be within a specific Scope, which corresponds to that property. This avoids degrading administration performance by minimising global communications while ensuring overall application coherence. This feature is especially relevant in large-scale and highly-dynamic systems.

The proposed Cube framework was detailed in-depth in chapter 5. Cube framework provides reusable and extensible support for specifying administration goals and runtime knowledge via formal modelling languages; and a set of Autonomic Managers which use these models to adapt the managed system. As

part of the Cube framework, the System Modelling Language (SML) defines four modelling layers with concepts ranging from generic to domain-specific and runtime elements. The most important layer defines Cube's meta-model concepts, which enables users to model managed system elements and integrate them into the Cube framework. Part of the Cube framework prototype, we have defined a set of domain-specific modelled elements for the data-mediation domain, which was targeted in this thesis. We have also proposed and developed a Goals Description Language (GDL), which allows users to specify administration objectives on the modelled elements via a graph-based syntax. The resulting description, called Archetype, is used by a set of identical Autonomic Managers to ensure the specified objectives using a distributed constraints-resolution algorithm.

With respect to Cube's Autonomic Manager design, most of its internal modules are made extensible so as to facilitate the adoption and customisation of the Cube framework across various targeted domains and applications. The design and implementation of the proposed Cube framework was detailed in chapter 6. The provided description covers the runtime infrastructure, extensions and utilities. We have also detailed the implementation choices - based mainly on standards like OSGi and XML - and explained the main Java classes of the implemented prototype.

Cube framework and its prototype implementation were evaluated in chapter 7. The Cube prototype was employed to provide autonomic features in the context of two national research projects: ANR Self-XL and MINALOGIC Medical. Within the first project, we conducted a quantitative and qualitative evaluation by monitoring Cube framework's operations when self-creating a distributed data-mediation application. We showed the framework's capability of instantiating and configuring the targeted application automatically, on an increasing number of distributed platforms. First, rather unsurprisingly, the measures confirmed that carrying-out such complex management operations via the framework will necessitate only a limited time when compared to manual or ad-hoc procedures. Second, more importantly, results indicated the way in which the Cube framework would scale with the increasing number of managed platforms and application components. Within the second project, we have conducted a more extended evaluation of our framework, both functional and qualitative, via a use case involving different self-management operations. These experiments highlighted Cube framework's capability of performing self-optimisation, self-healing and self-extension operations at runtime. Overall, the obtained results indicated Cube framework's viability for managing distributed data mediation systems while supporting important software engineering quality attributes.

8.2 Summary of thesis contributions

This thesis developed a solution for overcoming the limitations identified in the state-of-art of engineering autonomic computing systems. The main objective was to improve and to facilitate the development, execution and maintenance of autonomic systems so as to ensure self-management capabilities. Our main contributions consist of:

1. Providing an **autonomic framework** – called Cube framework - that supports a range of *quality attributes* and is characterised by:
   (a) A high-level, generic and graph-oriented *language* for specifying administration objectives (i.e., Archetype specification language);
   (b) A decentralised, modular and extensible *runtime* platform that ensures the specified objectives (i.e., Cube Autonomic Managers).
2. Identifying and applying several **techniques** in a synergistic manner in order to ensure important quality attributes in autonomic management. These techniques include: *architecture-oriented mod-*

elling, *constraint resolution*, *self-organisation* and *control theory*. Merged together, they enable a wide range of dynamic system adaptations while ensuring the achievement of predefined goals. More precisely, the framework's core resolver implements the constraints satisfaction paradigm to solve problems related to differences detected between the system's monitored state and the system's targeted state space, where state is represented in terms of system architecture. The resolver relies on system knowledge and domain-specific values, which are represented via Runtime Model instances that provide an abstracted description of the managed system. The framework is decentralised, with the Runtime Model being split and distributed across several Autonomic Managers. Each Autonomic Manager contains a resolver instance and controls a local part of the managed system. The Autonomic Managers self-organise into user-defined Scopes, controlled by local Scope Leaders, in order to minimise coordination messages and hence improve performance and scalability. Overall, the Cube self-management system represents an external, decentralised and architecture-based control loop.

3. A working **open-source tool set** that integrates: (1) a dedicated XML grammar and parser for user goals, respecting the proposed Archetype specification; (2) a lightweight, dynamically adaptable runtime platform based on OSGi standards; (3) a modular framework design and implementation enabling the seamless development and integration of customised domain-specific extensions, based on the Apache Felix iPOJO component model; and (4) a set of common extensions and tools helping debugging and monitoring Cube systems.
4. A concrete **framework application to distributed data mediation systems** to self-manage the instantiation and configuration of mediation components, their local and remote connections, the self-healing of distributed mediation chains and the self-optimisation of resource consumptions.

8.3 Limitations and perspectives

In the following sub-sections, we discuss the potential limitations of our framework, as well as the possible solutions for addressing them in future work. While some limitations are inherent to the Cube approach, others stem from the Cube framework that we propose. Since our contribution focuses on the Cube framework, in this section we only discuss limitations related to the framework and to its prototype implementation. These limitations can be grouped into two categories: those related to the Archetype specification; and those related to the Autonomic Manager's design and implementation. We describe each limitation and discuss possible solutions and enhancements to be developed in future research work.

8.3.1 Archetype Specification

The proposed Goal Description Language (GDL) - also called Archetype Specification Language - relies on a simple, yet powerful graph-based syntax. It allows administrators to specify their goals in a relatively natural manner, based on domain-specific descriptions of managed elements, their relationships and desired properties (see section 5.3 of chapter 5). Nevertheless, the proposed language (GDL) has some limitations that we discuss in this section.

Cardinality of Description Properties and of Goal Properties. The GDL only supports unary or binary Description Properties and Goal Properties. When an administrator wants to specify goals that imply three or more managed elements at the same time, this is not naturally supported by the Archetype's syntax and its runtime resolver. To address such cases, users should decouple the description of the managed system

part into binary relations and then associate these relations together. At the same time, the current Archetype Specification Language suffices for cases where goals and descriptions between managed elements are either unary (concerning managed element properties) or binary (concerning relations between two managed elements).

Goal expressivity. Based on the current GDL syntax, all the goals specified in the Archetype are mandatory. If one of the goals specified for a managed element is not resolved, then the managed element itself remains invalid. In the latest version of the prototype, we have added an "optional" attribute to the Archetype's Goal Description syntax. If this attribute is set to 'true' and no solution is found for the corresponding goal, then the subject managed element is still set to be valid (of course, if all the other mandatory goals are resolved). As future work, we have planned to add the following features to the Archetype Goal Descriptions and the corresponding Resolver, in order to enhance expressivity and extend the solutions perimeter when expressing and resolving goals, respectively:

1. *Priorities*. The first enhancement is to allow users to set priorities for goals defined on the same subject element. This can help resolve conflict problems among several goals. If two goals have contradictory effects on the subject element, then the Archetype Resolver will only consider and perform the action of the goal described with the highest priority.
2. *Logical operators*. Complementary to the priorities feature above, logical operators can be specified on various sets of goals, including for instance binary operators like "and", "or", and "xor". This extension can widen the space of possible solutions by allowing various combinations of goals to be achieved for validating a subject element. This will increase the expressivity and flexibility of Archetype Goals by supporting alternative solution branches, which the Archetype Resolver can take when resolving managed elements. For instance, specifying an "or" operator on a set of goals for a certain element will imply that the element will be valid as soon as any of the goals in the set is achieved.

Archetype correctness. It is difficult to prove formally that the user-provided specification corresponds to the user's intention in terms of administrative goals. This is an important issue, and currently, the user can only simulate and test their Archetype specification in order to ensure that the resulting solution is correct. As future work, we envisage to first provide specific tools for helping the user to detect anomalies in their Archetype syntax; and then to detect further semantic anomalies. Syntactic analysis can detect problems related to the XML grammar used for the Archetype specification (described in 6.2). Currently, this verification is performed at runtime when the Autonomic Manager loads Archetype XML file. On the short-term, we aim to provide a reference XSD file for the Archetype XML syntax so that the user's Archetype XML files can be verified directly using XML checkers. Further on, we aim to provide more advanced tools that can detect more subtle anomalies in the user's Archetype files. This includes for instance checking that the descriptions used for defining a goal's subject or object are of the required type. On the long-term, we aim to provide more advanced tools that can inform the user about the possible solutions that can be obtained from an Archetype specification. This can be done in a simulation environment, and can provide some valuable initial insights on the behaviour and results of Cube Autonomic Managers when given various Archetypes.

8.3.2 Runtime Autonomic Management

In this section, we discuss Cube framework's limitations with respect to runtime autonomic management and we elaborate some perspectives and future work.

Runtime Modelling. Cube framework's architectural modelling oriented approach and its corresponding runtime provides several useful features (discussed in chapters 4 and 5) and some limitations, which we discuss in the following points:

1. *Management delays*. As described in chapter 4, our framework relies on the reference architecture for autonomic management proposed by IBM (i.e. the MAPE-K control loop), where the managed system is decoupled from the autonomic managers. This design choice separates the autonomic infrastructure from the targeted managed system. It can provide better support for the separate evolution of the two parts and make the self-management infrastructure reusable across multiple use cases. However, this design choice raises several issues and features some important limitations. Most importantly, loose-coupling introduces synchronisation-related uncertainties between the managed system and its runtime model representation, which is maintained by Autonomic Managers to represent the managed system's state. In particular, there is an important uncertainty in knowing when a problem has been detected and when a change has been effected successfully. Indeed, the system state can change and cause new problems while Autonomic Managers still work on resolving previous problems.
   In an alternative, more tightly-coupled self-management approach, where the adaptation logic is embedded with the managed system implementation (e.g. as component containers or via exception handling), problem detection and corresponding adaptation changes can be effected in a more timely manner. However, coupling control logic with managed resources becomes difficult when the targeted system consists of multiple, heterogeneous and distributed sub-systems. In the Cube framework, monitors build a local Runtime Model of each managed system part. At the same that an Archetype Resolver detects an anomaly in its Runtime Model and tries to find a solution, monitors can update the Runtime Model and possibly change the very instances that are already part of resolution process.
   To address this issue in the implemented prototype, the archetype resolution is a blocking process. Modelled instances referenced in the Resolution Graph are marked as "blocked" so that no updates can be performed on it. This prevents the Resolver process from leaving the system in an incoherent state. However, the inputs that the Resolver uses may be deprecated as the real monitored elements have already changed. This may lead to the Resolver's solution failing to be implemented in the actual system, hence prompting the Resolver to start over the resolution process.
   To minimise the impact of this limitation, we propose a future resolution algorithm with partial solutions. This means that the Archetype Resolver finds partial solutions which are applied to the monitored system immediately, before continuing the resolution process. When applying such partial changes, monitors will also observe the managed system and update the Runtime Model to reflect its most recent state. Hence, the Resolver can continue the resolution process based on updated state values. As another solution, we can use distributed lock services like Chubby [Bur06]. However, it relies mainly on a centralised master node managing all the locking logic which impact the scalability and the performance of the Cube system.

2. *No distributed roll-back and transactions support*. For each managed element, once a solution that meets all its constraints is found, the solution is set in place and not rolled-back to accommodate the resolution of another element. This "greedy" approach ensures the convergence of the Archetype resolution process, with or without a solution, as the application grows monotonically towards a complete solution and stops when the solution can be no longer refined. Yet, in some cases, the choices a Resolver makes for addressing an initial constraint may prevent another Resolver from finding a solution for a subsequent constraint, later on. In such cases, if several choices where

available for the initial constraint, the initial solution may be rolled-back and an alternative solution implemented instead, hence providing an opportunity for the subsequent constraint to be resolved. While this future extension would render the resolution process more flexible it will also raise more important convergence issues that will have to be addressed.
Finally, once a complete solution is found in the overall Runtime Model, it must be implemented in the managed system. This may imply several executor operations by distributed Autonomic Managers. Distributed transactions should be set in place to ensure the coherence of such execution operations. Applying distributed transactions is possible in cases where the number of the impacted Autonomic Managers is limited. Distributed transactions represent a challenging topic where research issues and technical problems are still open.

3. *Decentralised adaptation delays*. In the Cube framework, adaptation delays may cause synchronisation issues at two different levels: first, between an Autonomic Manager and its managed system part (as already discussed in the management delay problem above); and second, among several Autonomic Managers that collaborate to find distributed solutions. Each Autonomic Manager is responsible for a portion of the global Runtime Model representing the overall managed system. That is, the Runtime Model providing the knowledge for self-management decisions is split among different Autonomic Managers, and decisions are made locally or via collaborations with neighbour Autonomic Managers. Important synchronisation issues may occur as Autonomic Managers use information from remote Runtime Models that may be modified by other Autonomic Managers in the meantime. In the current implementation, to avoid having several Autonomic Managers modify the same Runtime Model elements at the same time, hence risking incoherent system states, all access to Runtime Model elements is protected by local locks; a single AM may hold the lock and modify the model element at any one time. While this solution may suffice for the current prototype and experimented scenarios, the problem of synchronisation among decentralised processes represents a great challenge, subject to dedicated research projects. Existing solutions can be identified, adopted and integrated into the Cube framework in the future.

Static versus Dynamic Goals. When the Archetype is deployed onto the production infrastructure, the current framework prototype does not support any dynamic updating of the goals specified in Archetype; the Archetype is static. This is why users should specify goals as abstract as possible to support a large range of possible variations respecting the same Archetype. Introducing, modifying or removing Archetype goals and descriptions dynamically currently requires restarting the entire Cube system (i.e., all participating Cube Autonomic Managers). Having the possibility to change an Archetype dynamically remains a perspective of this thesis work. Different solutions can be explored to this aim. The first one is to start applying changes to Archetypes in the Autonomic Managers with no or very few external references to other Autonomic Managers; then progressively extend the list of updated AM. The second solution consists in building a new local Runtime Model that respects the new Archetype version, in parallel with the existing Runtime Model respecting the old Archetype. When all the instances in the new Runtime Model are resolved, we switch all the monitors/executors to handle these new instances; and discard to old Runtime Model.

Exploring alternative coordination strategies. In Cube framework and its prototype implementation, the coordination strategy relies on a collaboration among the Autonomic Managers involved in the resolution process; there is no competition among Autonomic Managers. As described in sub-section 4.4.3, the self-organisation of the Cube framework relies on three main techniques: (1) shared Archetype, (2) local, direct coordination between neighbouring Autonomic Managers to ensure the same goal; and (3)

semi-centralised coordination of Autonomic Managers that execute in well-defined Scopes for ensuring properties that must hold within those Scopes. We have shown the benefits of this coordination strategy to self-manage data-mediation systems. The applicability and efficiency of this strategy has to be studied and evaluated for other types of managed systems.

Alternative coordination strategies, based on hybrid or completely decentralised approaches, could be considered. This will affect the communication and interaction protocol in the Cube system (as described below), as well as the current implementation of the Archetype Resolver. Nevertheless, the modular and extensible architecture of Cube framework's Autonomic Manager will allow integrating new Archetype Resolvers and Communicators without affecting the other modules and extensions used in the current Autonomic Manager. Users may even be allowed to choose among several coordination strategies (e.g. via dedicated configurations in the Archetype) so as to best address the challenges of their specific managed systems.

In the context of data centres, we envision to study the effective integration of Apache ZooKeeper[1] coordination system [HKJR10] with Cube framework to self-manage multi-host and multi-process distributed C or Java based systems. Zookeeper is a high available and reliable coordination system. It can be used for leader election, group membership, and configuration maintenance through a shared hierarchical name space that is modelled after a file system. A possible integration strategy with Cube is to implement the Runtime Model and Communicator modules using Zookeeper, this can also impact the Archetype Resolver's current implementation. As in the current implemented Cube prototype, the data is kept in memory, in addition in Zookeeper the data is backed up to a log for reliability. By using memory Zookeeper is very fast and can handle the high loads across huge numbers of Autonomic Managers. Furthermore, Zookeeper nodes are organised in hierarchical organisation which allow adding nodes at any point of a tree, get a list of entries in a tree, get the value associated with an entry, and get notification of when an entry changes or goes away. All this features are essential to build new reliable version of the Cube prototype.

Exploring alternative communication and interaction protocols. The design of Cube Autonomic Managers considers a decentralised organisation of the Cube system in its different internal modules. For instance, the Runtime Model only contains modelled element instances of system elements that are managed locally. References to remote element instances are represented in the local Runtime Model based on the remote instance's universal identifier. Several internal modules where introduced to ensure the connectivity and the coordination between Autonomic Managers. This includes the Communicator and Life Controller modules. Communicator provides a communication language centred on the use of `CMessage` objects. The communication initiation entities (source AMs) specify the type of message (e.g., runtime model queries or archetype resolution procedures) and the supplementary information needed to accomplish the requested action by the communication target (receiver AM). Furthermore, the Communicator implementation defines the interaction protocol used between the different Autonomic Managers. The current Communicator implements an asynchronous request/reply protocol between AMs, based on direct interconnections (TCP/IP sockets). Other interaction protocols could be implemented and introduced into the Cube system. This include for instance the Publish/Subscribe protocol, the Dutch auction protocol [FIP01] (also known as 'public bid auction'), or the contract net protocol [Smi80]. The chosen interaction protocol will depend on the nature of the managed system and its administrative goals, and the implementation of the Archetype Resolver (as described in the previous point).

Resolution process. The current Archetype Resolver is based on a Constraint-based programming ap-

[1] http://zookeeper.apache.org/

proach to find solutions to detected problems. The corresponding graph-based algorithm we proposed was detailed in sub-section 5.4.3. It maps Archetype concepts into Constraint Satisfaction Problem (CSP) concepts. The domain value of each variable in a CSP represents corresponding runtime model instances, in either the local Autonomic Manager or in remote managers. In the following, we discuss some of the characteristics and possible enhancements of the current implementation.

1. The resolution process was designed and implemented to fit the specific requirements of the Cube framework. However, other constraint resolution engines can be adopted to resolve Cube Runtime Model instances with respect to user-provided Archetype specifications. To this aim, the adopted resolver should support user-specific objects (which are Runtime Model instances in our case) as domain values for CSP variables. We notice that the majority of constraint resolvers only support numerical values. In addition, distributed CSP resolvers would be best fitted to our Cube framework so as to follow the decentralised Cube approach.
2. Applications based on the presented resolution process conform to a fixed Archetype (template) and support flexible Archetype instantiations (context-sensitive solutions determined by the Archetype Resolvers of Cube Autonomic Managers). This resolution strategy offers a compromise between controlling essential application properties, and enabling application survival and adaptation in changing runtime contexts. Autonomic Managers simultaneously and progressively find and create flexible application solutions. Autonomic Managers do not have to determine when a complete solution is found before instantiating it. Also they will not attempt to find globally optimal solutions or to ensure 100% availability for managed systems. The main target is long-term survivability, adaptability and viability.
3. With respect to the decidability of the developed constraint resolution process, the core Archetype Resolver and Specific Resolvers provided as extensions are decidable if the number of available solutions for each constraint is finite (e.g. a limited number of implementations can be found for a component type). The resolution process sequentially addresses each constraint that applies to a managed element, using a backtracking process to progressively build and verify the solutions' tree. Each set of constraints is decidable if its solution tree is finite. For each managed element being resolved, once a solution that meets all its constraints is found, the solution is set in place and not rolled-back to accommodate another element's resolution. This "greedy" approach (as discussed before) ensures the convergence of the archetype resolution process, as the application grows monotonically towards a complete solution. Regarding the stability of the resulting application, this aspect will have to be studied depending on the context-sensitive adaptation of the Specific Resolvers included - this is a "classic" problem in autonomic and control systems. Further research can be carried out for exploring such resolver characteristics (convergence, stability and decidability) in various types of management scenarios.

8.3.3 Towards a reusable methodology and integrated development environment for engineering autonomic computing systems

In this thesis, we have proposed an autonomic computing framework to help engineering self-management systems. While we have concentrated on providing a framework design that would ensure several quality attributes, the proposed framework and the achievements we have accomplished opens good perspective and motivated us to look forward to define new software engineering autonomic computing methodology based on Cube framework.

The Cube framework already provides the following features:

1. Well defined autonomic system *Life-Cycle*. This aspect is divided into offline and online activities. Offline activities consist of providing model abstractions, implementing application-specific extensions (i.e. monitors, executors or resolvers) and writing goal specifications (Archetype file). Online activities consist in instantiating the different Autonomic Managers that use the shared Archetype, resolve problems related to the system's architecture, and possibly inform administrators about situations where no solution is found.
2. *Modelling framework.* We have proposed a modelling layer allowing the introduction of user-specific model abstractions and their seamless integration into the Cube system. In addition, the Goal Description Language, which uses the modelling framework to define goals for modelled managed elements, is also extensible and can be personalised to provide *domain-specific languages* (DSL) for final users that are experts in some specific technologies. The proposed XML syntax makes the goal specification as simple as possible using declarative constructs.
3. Well-defined *actor roles* (see Figure 8.1). There are three kinds of actor roles in the Cube system. First, technology experts are specialised in managing a specific technology of the global managed systems. They participate, together with Cube developers and Cube administrators, to writing the Archetype file. The Archetype corresponds to global administration goals whatever the underlying managed sub-systems. Second, Cube developers are responsible for implementing the various Cube extensions necessary for each specific application of business domain. Third, Cube administrators are responsible for deploying the Archetype and supervising the executing Cube system.
4. A Framework Runtime featuring a *modular* and *extensible* architecture and supporting important quality attributes, like scalability or dynamic adaptability.

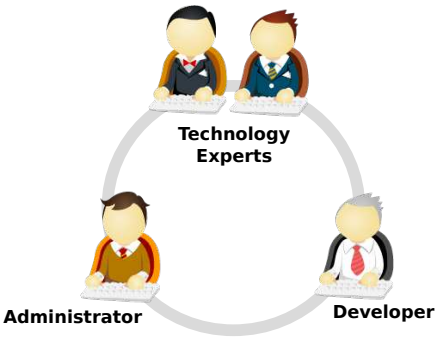


Figure 8.1: Actor roles

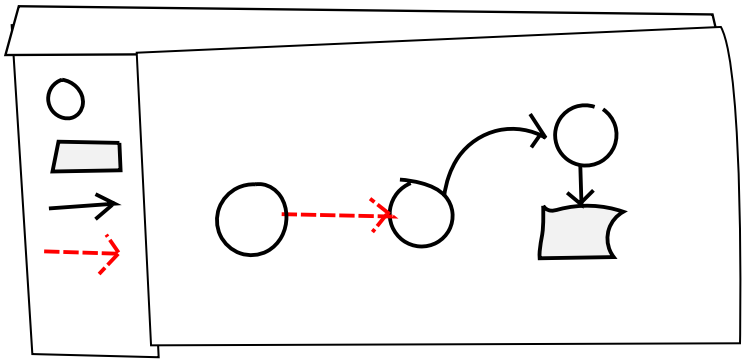
These features should be accompanied by other important aspects as listed in the following points:

1. *Integrated Development and Design Environment*: we should provide graphical tools helping the development tasks related to the Cube framework. There are two important features that should be provided in this respect:
   (a) Automatic code generator. Automatically produce the corresponding Java code for domain-specific managed elements.
   (b) Graphical Archetype Editor. Facilitate the Archetype writing. Research works in model-based editors has reached some maturity. Our Modelling framework is based on well-used modelling tools like EMF and its ECORE meta-model (chapter 5). We could equally benefit from other Eclipse modelling tools, such as GMF, to build our dedicated Graphical Editor. The Editor will

generate or edit the equivalent XML file automatically based on the graphical representation.

2. *Simulation and testing environment*. Providing a suitable simulation environment will help Cube actors to write correct Archetype specifications. However, evaluating autonomic system is difficult and remains an open challenging issue. A dedicated workshop is organised during the SASO conference to discuss and find solutions to this challenge.
3. Exploration of other domains. We should further experiment with the Cube system for self-managing other type of systems, like pervasive or cloud infrastructures. This will help reveal further limitations of the proposed framework and/or indicate its suitability for covering a large range of modern software systems. Our extensible framework can provide abstract modelling concepts for such systems, and provide the necessary mappings to the managed system resources and corresponding technologies.

Ultimate future objective. Our ultimate objective is to render the management of modern complex computing systems as simple as designing circles, rectangles and arrows. The Cube system will perform all the underlying self-management work in a reliable and effective way.



Appendix A

XML Terminology and Vocabulary

The following table gives a short summary about the XML terminology and vocabulary used in this thesis.

Term	**Short Definition**
Tag	A markup label that begins with < and ends with > and which indicates the start or the end of an element – but not the element content itself.
Element	A logical XML component which either begins with a start-tag (e.g., `<component>`) and ends with a matching end-tag (e.g., `</component>`) or consists only of an empty-element tag (e.g., `<component/>`).
Root Element	A well-formed XML document should have a single "root" element that contains all the other elements.
Child Element	An XML element that is contained within another element.
Parent Element	An XML element that contains other elements.
Attribute	A part of an element that provides additional information about that element (e.g., `<component type="Decoder"/>`).
XML namespace	A collection of names, identified by an URI reference, which is used in XML documents as element types (i.e. tags) and attribute names. (e.g., `<cube xmlns:core="fr.liglab.adele.cube.core"/>`)
Qualified Names	Provide a mechanism for concisely identifying an XML element or an attribute by a pair {URI, local-name} (e.g., `<core:component/>`).
Nesting	Placing one element inside another. When two tags are opened, they must be closed in the reverse order.

Appendix B

Implementation Details

This appendix lists some implementation details of the Cube framework.

B.1 Administration Service

The following figure shows the UML class diagram of the Administration Service interface. This interface is implemented by the `CubeRuntime` Java class presented in chapter 6.

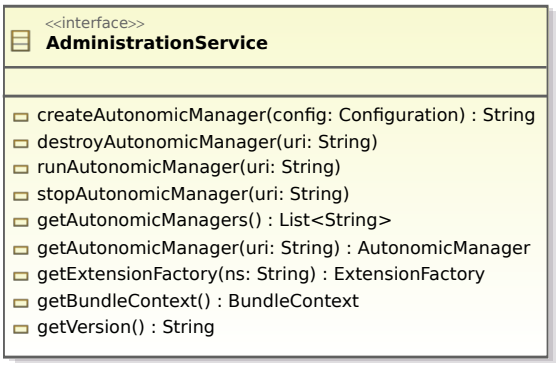


Figure B.1: Administration Service Java Interface

In the following, we provide some more detailed description of each of its methods.

- `String createAutonomicManager(Configuration config)` This method allows to instantiate a new Autonomic Manager with the provided configuration. The initial configuration is provided via a Configuration object which contains the initial setting information for the Autonomic Manager to be created, as well as the set of the initial extensions and their configurations. This class throws an `AutonomicManagerException` exception when the configuration is invalid. This is the case for instance when the provided host and port combination is already in use by another Autonomic Manager or another process. In the case when the instantiation process executes correctly, this method returns the URI of the newly created Autonomic Manager.
- `void destroyAutonomicManager(String uri)` This method allows the destruction and removal of a running Autonomic Manager identified by its URI. This method calls the stopAutonomicManager method to ensure the correct stopping of the Autonomic Manager before its destruction.

- `void runAutonomicManager(String uri)` This method allows running stopped Autonomic Managers.
- `void stopAutonomicManager(String uri)` This method allows stopping an Autonomic Manager momentarily. When stopped, none of the Autonomic Manager's internal modules or extension remains running; they are all stopped as well. Hence, the Autonomic Manager will not receive any messages from other Autonomic Managers. Stopping an Autonomic Manager will also free all its Runtime Model instances; all the related Autonomic Managers are notified about that.
- `List<String> getAutonomicManagers()` It returns the list of Autonomic Managers running on the local OSGi platform. The returned list contains the Autonomic Managers' URIs.
- `AutonomicManager getAutonomicManager(String uri)` It returns the AutonomicManager object having the given URI. If no Autonomic Manager is found, this method returns null.
- `ExtensionFactory getExtensionFactory(String namespace)` It returns the ExtensionFactory object having the given namespace. More precisely, when an Autonomic Manager is created, it gets access to the list of its initial extensions via the Configuration object provided as parameter by the user. For each extension's namespace, it finds the adequate `ExtensionFactory` provided as a service and creates a new extension instance. The extension instance will then be used within the Autonomic Manager.
- `BundleContext getBundleContext()` It returns the `BundleContext` object of the OSGi platform. This object provides access to bundle relation information.
- `String getVersion()` It returns the version of the current Cube implementation.

B.2 Component modelled element

The following listing shows the implementation detail of the "component" modelled element.

Listing B.1: Java class of "Componentt" modelled element

```
public class Component extends ManagedElement {

    public static final String NAME = "Component";
    public static final String CORE_COMPONENT_ID = "id";
    public static final String CORE_COMPONENT_TYPE = "type";
    public static final String CORE_COMPONENT_NODE = "node";
    public static final String CORE_COMPONENT_INPUTS = "inputs";
    public static final String CORE_COMPONENT_OUTPUTS = "outputs";

    public Component(String amUri) {
        super(amUri);
    }

    public Component(String amUri, Properties properties) throws
        PropertyExistException, InvalidNameException {
        super(amUri, properties);
        setNamespace(CoreExtensionFactory.NAMESPACE);
        setName(NAME);
    }

    public void setComponentId(String component_identifier) {
```

```
21          try {
22              if (this.getAttribute(CORE_COMPONENT_ID) == null)
23                  this.addAttribute(CORE_COMPONENT_ID, component_identifier);
24              else
25                  this.updateAttribute(CORE_COMPONENT_ID, component_identifier);
26          } catch (PropertyNotExistException e) {
27              e.printStackTrace();
28          } catch (InvalidNameException e) {
29              e.printStackTrace();
30          } catch (PropertyExistException e) {
31              e.printStackTrace();
32          }
33      }
34
35      public String getComponentId() {
36          return this.getAttribute(CORE_COMPONENT_ID);
37      }
38
39      public void setComponentType(String component_type) {
40          try {
41              if (this.getAttribute(CORE_COMPONENT_TYPE) == null)
42                  this.addAttribute(CORE_COMPONENT_TYPE, component_type);
43              else
44                  this.updateAttribute(CORE_COMPONENT_TYPE, component_type);
45          } catch (PropertyNotExistException e) {
46              e.printStackTrace();
47          } catch (InvalidNameException e) {
48              e.printStackTrace();
49          } catch (PropertyExistException e) {
50              e.printStackTrace();
51          }
52      }
53
54      public String getComponentType() {
55          return this.getAttribute(CORE_COMPONENT_TYPE);
56      }
57
58      public void setNode(String node_url) {
59          Reference r = null;
60          try {
61              r = this.addReference(CORE_COMPONENT_NODE, true);
62          } catch (InvalidNameException e) {
63              e.printStackTrace();
64          }
65          r.addReferencedElement(node_url);
66      }
67
68      public String getNode() {
69          Reference r = getReference(CORE_COMPONENT_NODE);
70          if (r != null && r.getReferencedElements().size() > 0) {
71              return r.getReferencedElements().get(0);
72          }
73          return null;
74      }
75
76      public boolean addInputComponent(String compURI) {
77          Reference r = null;
```

```
        try {
            r = addReference(CORE_COMPONENT_INPUTS, false);
        } catch (InvalidNameException e) {
            e.printStackTrace();
        }
        return r.addReferencedElement(compURI);
    }

    public List<String> getInputComponents() {
        Reference r = this.getReference(CORE_COMPONENT_INPUTS);
        if (r != null) {
            return r.getReferencedElements();
        }
        return new ArrayList<String>();
    }

    public boolean addOutputComponent(String compURI) {
        Reference r = null;
        try {
            r = addReference(CORE_COMPONENT_OUTPUTS, false);
        } catch (InvalidNameException e) {
            e.printStackTrace();
        }
        return r.addReferencedElement(compURI);
    }

    public List<String> getOutputComponents() {
        Reference r = this.getReference(CORE_COMPONENT_OUTPUTS);
        if (r != null) {
            return r.getReferencedElements();
        }
        return new ArrayList<String>();
    }
}
```

B.3 Connected Specific Resolver

The following listing shows the implementation detail of the “connected” specific resolver.

Listing B.2: Java class of the "connected" specific resolver

```
public class Connected extends AbstractResolver {

    public Connected(Extension extension) {
        super(extension);
    }

    public String getName() {
        return this.getClass().getSimpleName();
    }

    public boolean check(ManagedElement me, String value) {
        if (me != null && value != null) {
            Reference r = me.getReference(Component.CORE_COMPONENT_OUTPUTS);
            if (r!=null) {
```

```
15                  for (String e : r.getReferencedElements()) {
16                      if (e.equalsIgnoreCase(value)) return true;
17                  }
18              }
19          }
20          return false;
21      }
22
23      public boolean perform(ManagedElement me, String value) {
24          if (me != null && value != null) {
25              Reference r = me.getReference(Component.CORE_COMPONENT_OUTPUTS);
26              if (r == null) {
27                  try {
28                      r = me.addReference(Component.CORE_COMPONENT_OUTPUTS, false);
29                      RuntimeModelController rmc = getExtension().getAutonomicManager()
                            .getRuntimeModelController();
30                      rmc.addReferencedElement(value, Component.CORE_COMPONENT_INPUTS,
                            me.getUUID());
31                      r.addReferencedElement(value);
32                  } catch (InvalidNameException e) {
33                      e.printStackTrace();
34                  }
35              } else {
36              r.addReferencedElement(value);
37              }
38              return true;
39          }
40          return false;
41      }
42
43      public List<String> find(ManagedElement me, ManagedElement description) {
44          List<String> result = new ArrayList<String>();
45          if (me != null) {
46              Reference r = me.getReference(Component.CORE_COMPONENT_INPUTS);
47              if (r != null) {
48                  for (String tmp : r.getReferencedElements()) {
49                      ManagedElement input = getExtension().getAutonomicManager().
                            getRuntimeModelController().getCopyOfManagedElement(tmp);
50                      if (ModelUtils.compareTwoManagedElements(description, input) ==
                            0) {
51                          result.add(tmp);
52                      }
53                  }
54              }
55          }
56          return result;
57      }
58  }
```

B.4 Console Commands

Table B.1: Cube Console Commands

Command name	Format	Short Description
cube:version	$ version	Shows the Cube Version
cube:ams	$ ams	Shows the Autonomic Managers executing locally. They are identified by incrementing integers which can then be used by other commands to uniquely identify the Autonomic Manager. We refer to this AM identifier number by [AM]
cube:extensions	$ extensions [AM]	Shows the list of extensions associated to the given Autonomic Manager
cube:addext	$ addext [AM] [EXT_ID]	Dynamically adds an extension (identified by EXT_ID) to an Autonomic Manager (identified by AM)
cube:rmext	$ rmext [AM] [EXT_ID]	Dynamically removes an extension from an Autonomic Manager
cube:arch	$ arch [AM]	Shows the Archetype document of a given Autonomic Manager in XML format
cube:rm	$ rm [AM]	Shows the Local Runtime Model instances
cube:newi	$ newi [AM] [DESC]	Creates a new modelled element instance whose description is provided in the Archetype and identified by [DESC]
cube:update	$ update [AM] [INST_URI] [PNAME=PVALUE]	Updates the properties and/or the references of a given Runtime Model instance
cube:rmi	$ rmi [AM] [INST_URI]	Removes a modelled element instance from the Runtime Model of a given Autonomic Manager

Appendix C

Archetype XML files

This appendix lists the two Archetype XML files used in the evaluation chapter.

Listing C.1: Archetype file of the first case study

```
<?xml version="1.0"?>
<cube xmlns:core="fr.liglab.adele.cube.core" cube-version="2.0">
  <archetype id="net.debbabi.cube.uc1" version="1.0">
    <!-- ====== ADMINISTRATION GOALS ====== -->
    <goals>
      <goal>
        <inScope s="@datacentre" o="@central" r="fc"/>
        <inScope s="@server" o="@city" r="fc"/>
        <inScope s="@gateway" o="@city" r="f"/>
        <holdComponent s="@gateway" o="@hc" r="fc"/>
        <holdComponent s="@server" o="@rc" r="fc"/>
        <holdComponent s="@datacentre" o="@na" r="fc"/>
      </goal>
      <goal>
        <connected s="@gp" o="@hc" r="f"/>
        <connected s="@ep" o="@hc" r="f"/>
        <connected s="@wp" o="@hc" r="f"/>
        <connected s="@hc" o="@city_rc" r="f"/>
      </goal>
      <goal>
        <connected s="@rc" o="@city_cc" r="fc"/>
        <connected s="@cc" o="@central_na" r="f"/>
      </goal>
    </goals>

    <elements>
      <!-- ====== MASTER and SCOPE DESCRIPTIONS ====== -->
      <master id="master">
        <locatedIn o="cube://localhost:19000"/>
      </master>
      <scope id="central">
        <hasScopeId o="central"/>
        <controlledBy o="@master"/>
      </scope>
      <scope id="city">
        <hasScopeId o="${city}"/>
        <controlledBy o="@master"/>
      </scope>
```

```
39        <!-- ============ NODE DESCRIPTIONS ============ -->
40        <node id="datacentre">
41          <hasNodeType o="Datacentre"/>
42        </node>
43        <node id="server">
44          <hasNodeType o="Server"/>
45        </node>
46        <node id="gateway">
47          <hasNodeType o="Gateway"/>
48        </node>
49
50        <node id="city_server">
51          <hasNodeType o="Server"/>
52          <inScope o="@city"/>
53        </node>
54        <node id="central_datacentre">
55          <hasNodeType o="Datacentre"/>
56          <inScope o="@central"/>
57        </node>
58        <!-- ============ COMPONENT DESCRIPTIONS ============ -->
59        <component id="hc">
60          <hasComponentType o="HC"/>
61        </component>
62        <component id="rc">
63          <hasComponentType o="RC"/>
64        </component>
65        <component id="na">
66          <hasComponentType o="NA"/>
67        </component>
68        <component id="gp">
69          <hasComponentType o="GP"/>
70        </component>
71        <component id="ep">
72          <hasComponentType o="EP"/>
73        </component>
74        <component id="wp">
75          <hasComponentType o="WP"/>
76        </component>
77
78        <component id="city_rc">
79          <hasComponentType o="RC"/>
80          <onNode o="@city_server"/>
81          <HasMaxInputComponents o="5"/>
82        </component>
83
84        <component id="cc">
85          <hasComponentType o="CC"/>
86        </component>
87        <component id="city_cc">
88          <hasComponentType o="CC"/>
89          <onNode o="@city_server"/>
90        </component>
91
92        <component id="central_na">
93          <hasComponentType o="NA"/>
94          <onNode o="@central_datacentre"/>
95        </component>
```

```
    </elements>
  </archetype>
</cube>
```

Listing C.2: Archetype file of the second case study

```
<?xml version="1.0"?>
<cube xmlns:core="fr.liglab.adele.cube.core" cube-version="2.0">
  <archetype id="net.debbabi.cube.uc2" version="1.0">
    <!-- ====== ADMINISTRATION GOALS ====== -->
    <goals>
      <goal>
        <inScope s="@agg_server" o="@agg_cluster" r="fc"/>
        <inScope s="@med_server" o="@med_cluster" r="fc"/>
      </goal>
      <goal>
        <holdComponent s="@agg_server" o="@wsprobe" r="c"/>
        <connected s="@wsprobe" o="@loadbalancer" r="fc"/>
      </goal>
      <goal>
        <holdComponent s="@active_med_server" o="@ahd_box" r="c"/>
        <hasSourceComponent s="@ahd-box" o="@agg_loadbalancer" r="f"/>
        <connected s="@ahd-box" o="@hl7-router" r="fc"/>
        <connected s="@hl7-router" o="@hl7-validator" r="fc"/>
        <connected s="@hl7-validator" o="@validator" r="fc"/>
        <connected s="@validator" o="@translator" r="fc"/>
      </goal>
      <goal>
        <hasSourceComponent s="@phr-integration" o="@translator2" r="f"/>
      </goal>
      <goal>
        <hasNoComponents s="@inactive_med_server" o="all"/>
      </goal>
    </goals>

    <elements>
      <!-- ====== MASTER and SCOPE DESCRIPTIONS ====== -->
      <master id="master">
        <locatedIn o="cube://localhost:38000"/>
      </master>
      <scope id="agg_cluster">
        <hasScopeId o="AggregationCluster"/>
        <controlledBy o="@master"/>
      </scope>
      <scope id="med_cluster">
        <hasScopeId o="MediationCluster"/>
        <controlledBy o="@master"/>
      </scope>
      <!-- ============ NODE DESCRIPTIONS ============= -->
      <node id="agg_server">
        <hasNodeType o="AggregationServer"/>
      </node>
      <node id="med_server">
        <hasNodeType o="MediationServer"/>
      </node>
      <node id="active_med_server">
        <hasNodeType o="MediationServer"/>
        <hasProperty o="active:true"/>
```

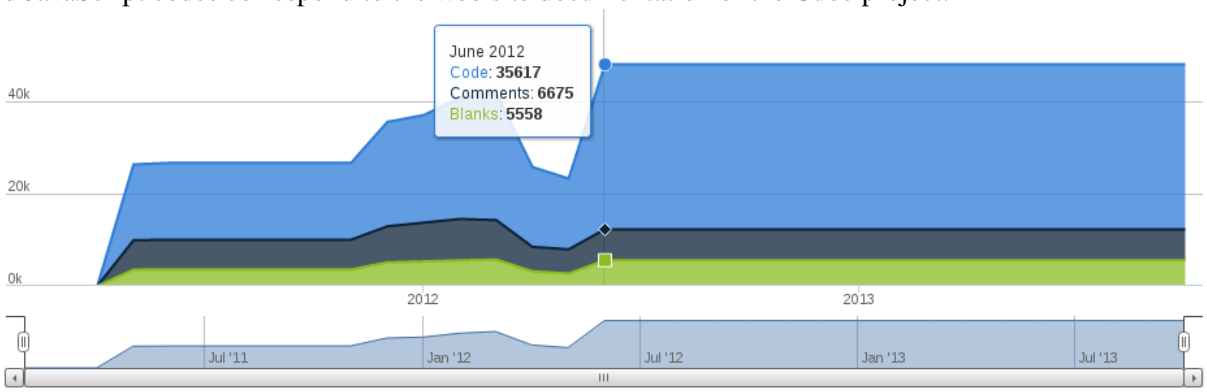
```
53        </node>
54        <node id="lb_agg_server">
55          <hasNodeType o="AggregationServer"/>
56          <inScope o="@agg_cluster"/>
57        </node>
58        <node id="inactive_med_server">
59          <hasNodeType o="MediationServer"/>
60          <hasProperty o="active:false"/>
61        </node>
62        <!-- ============ COMPONENT DESCRIPTIONS ======== -->
63        <component id="wsprobe">
64          <hasComponentType o="WSProbe"/>
65          <hasWSSources o="file:///ws-endpoints.xml"/>
66        </component>
67        <component id="loadbalancer">
68          <hasComponentType o="LoadBalancer"/>
69          <onSameNode o="@wsprobe"/>
70          <hasDispatcher o="LoadBalancing"/>
71        </component>
72        <component id="ahd_box">
73          <hasComponentType o="AHD-Box"/>
74          <onNode o="@active_med_server"/>
75        </component>
76        <component id="agg_loadbalancer">
77          <hasComponentType o="LoadBalancer"/>
78          <onNode o="@lb_agg_server"/>
79        </component>
80        <component id="hl7-router">
81          <hasComponentType o="HL7-Router"/>
82          <onSameNode o="@ahd_box"/>
83        </component>
84        <component id="hl7-validator">
85          <hasComponentType o="HL7-Validator"/>
86          <onSameNode o="@hl7-router"/>
87        </component>
88        <component id="validator">
89          <hasComponentType o="Validator"/>
90          <onSameNode o="@hl7-validator"/>
91        </component>
92        <component id="translator">
93          <hasComponentType o="Translator"/>
94          <onSameNode o="@validator"/>
95          <hasDispatcher o="Multicast"/>
96        </component>
97        <component id="phr-integration">
98          <hasComponentType o="PHR-Integration"/>
99        </component>
100       <component id="@translator2">
101         <hasComponentType o="Translator"/>
102         <onSameNode o="@phr-integration"/>
103       </component>
104     </elements>
105   </archetype>
106 </cube>
```

Appendix D

Source code statistics

The source code was initially hosted using a service provided by the Grenoble University and managed using subversion[1]. The entire project was migrated to *github*[2] for more visibility and to facilitate external collaborations.

The following screen-shot is taken from *ohloh.net*[3] website which measures some metrics of open source projects. The present data is about the initial version of Cube (until June 2012). The HTML, CSS and JavaScript codes correspond to the web site documentation of the Cube project.



Language Breakdown

Language	Code Lines	Comment Lines	Comment Ratio	Blank Lines	Total Lines	Total Percentage
HTML	17,674	799	4.3%	2,641	21,114	44.1%
Java	10,073	5,257	34.3%	2,298	17,628	36.8%
XML	4,096	145	3.4%	183	4,424	9.2%
JavaScript	2,389	453	15.9%	174	3,016	6.3%
CSS	1,176	21	1.8%	224	1,421	3.0%
shell script	193	0	0.0%	38	231	0.5%
MetaFont	16	0	0.0%	0	16	0.0%
Totals	35,617	6,675		5,558	47,850	

[1] https://forge.imag.fr/projects/cube/
[2] https://github.com/AdeleResearchGroup/cube
[3] https://www.ohloh.net/p/cube/

Bibliography

[AAB+07] F. Akkawi, K. Akkawi, A. Bader, M. Ayyash, D. Fletcher, and K. Alzoubi, *Software adaptation: A conscious design for oblivious programmers*, Aerospace Conference, 2007 IEEE, march 2007, pp. 1 –12. 2.5.1

[AAC+00] H. Abelson, D. Allen, D. Coore, C. Hanson, G. Homsy, T.F. Knight Jr, R. Nagpal, E. Rauch, G.J. Sussman, and R. Weiss, *Amorphous computing*, Communications of the ACM **43** (2000), no. 5, 74–82. 3.3.2

[ACMS06] Ip Agarwala, Yuan Chen, Dejan Milojicic, and Karsten Schwan, *Qmon: Qos- and utility-aware monitoring in enterprise systems*, In The 3rd IEEE International Conference on Autonomic Computing, IEEE Computer Society, 2006, pp. 124–133. 2.4.3

[ADG98] Robert Allen, Remi Douence, and David Garlan, *Specifying and analyzing dynamic software architectures*, Proceedings of the 1998 Conference on Fundamental Approaches to Software Engineering (FASE'98) (Lisbon, Portugal), March 1998, An expanded version of a the paper "Specifying Dynamism in Software Architectures," which appeared in the Proceedings of the Workshop on Foundations of Component-Based Software Engineering, September 1997. 3.4.3

[AFG+04] Matthew Arnold, Stephen J. Fink, David Grove, Michael Hind, and Peter F. Sweeney, *A survey of adaptive optimization in virtual machines*, PROCEEDINGS OF THE IEEE, 93(2), 2005. SPECIAL ISSUE ON PROGRAM GENERATION, OPTIMIZATION, AND ADAPTATION, 2004. 3.1.2

[AH05] Nejla Amara-Hachmi, *A framework for building adaptive mobile agents*, Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems (New York, NY, USA), AAMAS '05, ACM, 2005, pp. 1369–1369. 3.2.3

[All07] OSGi Alliance, *Osgi service platform core specification, release 4.1*, `http://www.osgi.org/Specifications`, 2007. 6, 6.1.1.1

[Ant04] R.J. Anthony, *Emergence: a paradigm for robust and scalable distributed applications*, Autonomic Computing, 2004. Proceedings. International Conference on, may 2004, pp. 132 – 139. 3.3.1, 3.3.3, 4

[APS11] Richard Anthony, Mariusz Pelc, and Haffiz Shuaib, *The interoperability challenge for autonomic computing*, EMERGING 2011, The Third International Conference on Emerging Network Intelligence, 2011, pp. 13–19. 2.5.1, 5.4.3

[Ast07] Karl Johan Astrom, *Challenges in control education*, Proc. of the 7th IFAC Symposium on Advances in Control Education, Plenary Lecture, June, 2007, pp. 21–23. 3.1.3

[BBF09] G. Blair, N. Bencomo, and R.B. France, *Models@ run.time*, Computer **42** (2009),

no. 10, 22 –27. 3.4.3, 5.1.2

[BBH+05] Sara Bouchenak, Fabienne Boyer, Daniel Hagimont, Sacha Krakowiak, Adrian Mos, Noel De Palma, Vivien Quema, and Jean bernard Stefani, *Architecture-based autonomous repair management: An application to j2ee clusters*, In 24th IEEE Symposium on Reliable Distributed Systems (SRDS-2005, 2005, pp. 13–24. 2.5.2

[BBPLP98] Francois Bousquet, Innocent Bakam, Hubert Proton, and Christophe Le Page, *Cormas: common-pool resources and multi-agent systems*, Tasks and Methods in Applied Artificial Intelligence (1998), 826–837. 3.2.5

[BCDW04] J.S. Bradbury, J.R. Cordy, J. Dingel, and M. Wermelinger, *A survey of self-management in dynamic software architecture specifications*, Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems, ACM, 2004, p. 33. 2.5.2, 3.4.4

[BCK03] L. Bass, P. Clements, and R. Kazman, *Software architecture in practice*, Addison-Wesley Professional, 2003. 3.4.1, 3.4.3

[BCL+06] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani, *The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems*, Softw. Pract. Exper. **36** (2006), no. 11-12, 1257–1284. 2.5.2

[BCS09] Gordon Blair, Thierry Coupaye, and Jean-Bernard Stefani, *Component-based architecture: the fractal initiative*, Annals of Telecommunications **64** (2009), no. 1, 1–4. 3.1.2

[BDLM11] Johann Bourcier, Ada Diaconescu, Philippe Lalanda, and Julie A. McCann, *Autohome: An autonomic management framework for pervasive home applications*, ACM Trans. Auton. Adapt. Syst. **6** (2011), no. 1, 8:1–8:10. 2.5.2

[Bea11] Jacob Beal, *Functional blueprints: An approach to modularity in grown systems*, Swarm Intelligence **5** (2011), no. 3, 257–281. 3.3.2

[BFH03] Fran Berman, Geoffrey Fox, and Anthony JG Hey, *Grid computing: making the global infrastructure a reality*, vol. 2, Wiley, 2003. 1.1, 2.1.2

[BG01] Jean Bezivin and Olivier Gerbe, *Towards a precise definition of the omg/mda framework*, Proceedings of the 16th IEEE international conference on Automated software engineering (Washington, DC, USA), ASE '01, IEEE Computer Society, 2001, pp. 273–. 5.1.2

[BHW07] Rafael H Bordini, Jomi Fred Hübner, and Michael Wooldridge, *Programming multi-agent systems in agentspeak using jason*, vol. 8, Wiley-Interscience, 2007. 3.2.5

[BJM05a] Ozalp Babaoglu, Mark Jelasity, and Alberto Montresor, *Gossip-based self-managing services for large scale dynamic networks*, Service Management and Self-Organization in IP-based Networks (Dagstuhl, Germany) (Matthias Bossardt, Georg Carle, D. Hutchison, Hermann de Meer, and Bernhard Plattner, eds.), Dagstuhl Seminar Proceedings, no. 04411, Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2005. 4.4.3

[BJM+05b] Ozalp Babaoglu, Márk Jelasity, Alberto Montresor, Christof Fetzer, Stefano Leonardi, Aad van Moorsel, and Maarten van Steen, *Self-star properties in complex information systems: Conceptual and practical foundations (lecture notes in computer science)*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005. 2.3, 2.5.1

[BKPH09] Frances M. T. Brazier, Jeffrey O. Kephart, H. Van Dyke Parunak, and Michael N. Huhns, *Agents and service-oriented computing for autonomic computing: A research agenda*, IEEE Internet Computing **13** (2009), 82–87. 2.3, 3.2.4

[BMH06] Bill Burke and Richard Monson-Haefel, *Enterprise javabeans 3.0*, 5. ed., O'Reilly, Beijing, 2006. 1.1

[Bou08] Johann Bourcier, *Auto-home: une plate-forme pour la gestion autonomique d'applications pervasives*, Ph.D. thesis, University Joseph Fourrier, 2008. 2.5.2

[BPG+04] Paolo Bresciani, Anna Perini, Paolo Giorgini, Fausto Giunchiglia, and John Mylopoulos, *Tropos: An agent-oriented software development methodology*, Autonomous Agents and Multi-Agent Systems **8** (2004), no. 3, 203–236. 3.2.5

[BPHT06] S. Bouchenak, N. De Palma, D. Hagimont, and C. Taton, *Autonomic management of clustered applications*, Cluster Computing, IEEE International Conference on **0** (2006), 1–11. 2.5.2

[BPL04] Lars Braubach, Alexander Pokahr, and Winfried Lamersdorf, *Jadex: A short overview*, Main Conference Net. ObjectDays, vol. 2004, 2004, pp. 195–207. 3.2.5

[BPR01] Fabio Bellifemine, Agostino Poggi, and Giovanni Rimassa, *Jade: a fipa2000 compliant agent development environment*, Proceedings of the fifth international conference on Autonomous agents, ACM, 2001, pp. 216–217. 3.2.5

[BSP+02] J. P. Bigus, D. A. Schlosnagle, J. R. Pilgrim, W. N. Mills, and Y. Diao, *Able: a toolkit for building multiagent autonomic systems*, IBM Syst. J. **41** (2002), no. 3, 350–371. 3.2.4

[BTRC05] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés, *Automated reasoning on feature models*, Proceedings of the 17th international conference on Advanced Information Systems Engineering (Berlin, Heidelberg), CAiSE'05, Springer-Verlag, 2005, pp. 491–503. 3.4.3

[Bur06] Mike Burrows, *The chubby lock service for loosely-coupled distributed systems*, Proceedings of the 7th symposium on Operating systems design and implementation (Berkeley, CA, USA), OSDI '06, USENIX Association, 2006, pp. 335–350. 1

[Cap10] Bogdan Alexandru Caprarescu, *Robustness and scalability: a dual challenge for autonomic architectures*, Proceedings of the Fourth European Conference on Software Architecture: Companion Volume (New York, NY, USA), ECSA '10, ACM, 2010, pp. 22–26. 2.5.1, 3.3.3

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker, *Generative programming: methods, tools, and applications*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000. 3.4.3

[CGFP09] Carlos Cetina, Pau Giner, Joan Fons, and Vicente Pelechano, *Autonomic computing through reuse of variability models at runtime: The case of smart homes*, Computer **42** (2009), no. 10, 37–43. 3.4.3, 3.4.3, 5.1.2

[CGS+02] Shang-Wen Cheng, David Garlan, Bradley R. Schmerl, Joao Pedro Sousa, Bridget Spitnagel, and Peter Steenkiste, *Using architectural style as a basis for system self-repair*, Proceedings of the IFIP 17th World Computer Congress - TC2 Stream / 3rd IEEE/IFIP Conference on Software Architecture: System Design, Development and Maintenance (Deventer, The Netherlands, The Netherlands), WICSA 3, Kluwer, B.V.,

2002, pp. 45–59. 3.4.3

[CGS09] Shang-Wen Cheng, D. Garlan, and B. Schmerl, *Raide for engineering architecture-based self-adaptive systems*, Software Engineering - Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on, may 2009, pp. 435 –436. 2.5.2

[CH03] Humberto Cervantes and Richard S. Hall, *Automating service dependency management in a service-oriented component model*, Proceedings of the 6th Workshop of the European Software Engineering Conference/Foundations of Software EngineeringComponent Based Software Engineering, September 2003, pp. 379–382. 6.1.1.1

[CHW98] James Coplien, Daniel Hoffman, and David Weiss, *Commonality and variability in software engineering*, IEEE Softw. **15** (1998), no. 6, 37–45. 3.4.3

[CI09] Alain Cardon and Mhamed Itmi, *Multi agent modeling of self adaptive system for large scale complex systems*, Proceedings of the 2009 Summer Computer Simulation Conference (Vista, CA), SCSC '09, Society for Modeling & Simulation International, 2009, pp. 476–481. 3.2.4

[CN96] Paul C. Clements and Linda M. Northrop, *Software architecture: An executive overview*, 1996. 3.4.1

[CNNL98] Jaron C Collis, Divine T Ndumu, Hyacinth S Nwana, and Lyndon C Lee, *The zeus agent building tool-kit*, BT Technology Journal **16** (1998), no. 3, 60–68. 3.2.5

[Com00] Standards Committee, *Ieee recommended practice for architectural description of software-intensive systems*, October **1471-2000** (2000), no. 42010, i–23. 3.4.1

[CSWW04] D.M. Chess, A. Segal, I. Whalley, and S.R. White, *Unity: experiences with a prototype autonomic computing system*, Autonomic Computing, 2004. Proceedings. International Conference on, may 2004, pp. 140 – 147. 2.5.2, 3.1.2

[CZ04] Luca Cernuzzi and Franco Zambonelli, *Adaptive organizational changes in agent-oriented methodologies*, 2004. 3.2.1, 3.2.3

[CZ06] Luca Cernuzzi and Franco Zambonelli, *Dealing with adaptive multi-agent organizations in the gaia methodology*, Proceedings of the 6th international conference on Agent-Oriented Software Engineering (Berlin, Heidelberg), AOSE'05, Springer-Verlag, 2006, pp. 109–123. 3.2.1, 3.2.3

[DBE08] Ada Diaconescu, Johann Bourcier, and Clement Escoffier, *Autonomic ipojo: Towards self-managing middleware for ubiquitous systems*, Proceedings of the 2008 IEEE International Conference on Wireless & Mobile Computing, Networking & Communication (Washington, DC, USA), WIMOB '08, IEEE Computer Society, 2008, pp. 472–477. 3.1.2

[DDL12] Bassem Debbabi, Ada Diaconescu, and Philippe Lalanda, *Controlling self-organising software applications with archetypes*, Self-Adaptive and Self-Organizing Systems (SASO), 2012 IEEE Sixth International Conference on, 2012, pp. 69–78. 1.2, 4.2.1, 4.4.3, 7.2.1

[Deb09] Bassem Debbabi, *Framework orienté-service de médiation de données*, Tech. report, UFR IMAG Université de Grenoble, 2009. 1.4, 7.1.2

[DHP+05] Y. Diao, J.L. Hellerstein, Sujay Parekh, R. Griffith, G. Kaiser, and D. Phung, *Self-*

managing systems: a control theory foundation, Engineering of Computer-Based Systems, 2005. ECBS '05. 12th IEEE International Conference and Workshops on the, april 2005, pp. 441 – 448. 3.1.2

[DL09] A. Diaconescu and P. Lalanda, *A decentralized, architecture-based framework for self-growing applications*, Proceedings of the 6th international conference on Autonomic computing, ACM, 2009, pp. 55–56. 1.2, 4.2.1

[DL11] _____, *Self-growing applications from abstract architectures an application to data-mediation systems*, Evolving and Adaptive Intelligent Systems (EAIS), 2011 IEEE Workshop on, IEEE, 2011, pp. 170–177. 1.2, 4.2.1, 4.4.3

[DML08] Ada Diaconescu, Yoann Maurel, and Philippe Lalanda, *Autonomic management via dynamic combinations of reusable strategies*, Proceedings of the 2nd International Conference on Autonomic Computing and Communication Systems (ICST, Brussels, Belgium, Belgium), Autonomics '08, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008, pp. 16:1–16:10. 2.5.2

[Dou11] Rene Doursat, *Morphogenetic engineering weds bio self-organization to human-designed systems*, PerAda Magazine: Towards Pervasive Adaptation (2011). 3.3.2

[DPBB+08] Noel De Palma, Sara Bouchenak, Fabienne Boyer, Daniel Hagimont, Sylvain Sicard, and Christophe Taton, *Jade: un environnement d'administration autonome*, Technique et Science Informatiques **27** (2008), no. 9-10, 1225–1252. 2.5.2

[Dre07] F. Dressler, *Self-organization in sensor and actor networks*, Wiley series in communications networking & distributed systems, John Wiley & Sons, 2007. (document), 2.1.2, 3.8

[DRW06] Leticia Duboc, David S. Rosenblum, and Tony Wicks, *A framework for modelling and analysis of software systems scalability*, Proceedings of the 28th international conference on Software engineering (New York, NY, USA), ICSE '06, ACM, 2006, pp. 949–952. 2.5.1, 7.4

[DSC+00] Jim Dowling, Tilman Schäfer, Vinny Cahill, Peter Haraszti, and Barry Redmond, *Using reflection to support dynamic adaptation of system software: A case study driven evaluation*, Driven Evaluation?, OOPSLA Workshop on ObjectOriented Reflection and Software Engineering, 2000, pp. 9–9. 3.4.3

[DSM11] R Doursat, H Sayama, and O Michel, *Morphogenetic engineering: Toward programmable complex systems*, 2011. 3.3.2

[DSNH10] Simon Dobson, Roy Sterritt, Paddy Nixon, and Mike Hinchey, *Fulfilling the vision of autonomic computing*, Computer **43** (2010), no. 1, 35–41. (document), 2.1.1, 2.3, 4.1

[Dur99] Edmund H. Durfee, *Practically coordinating*, AI Magazine **20** (1999), no. 1, 99–116. 3.2.2

[DvdHT02] E.M. Dashofy, A. van der Hoek, and R.N. Taylor, *Towards architecture-based self-healing systems*, Proceedings of the first workshop on Self-healing systems, ACM, 2002, pp. 21–26. 3.4.2

[DWSHR05] Tom De Wolf, Giovanni Samaey, Tom Holvoet, and Dirk Roose, *Decentralised autonomic computing: Analysing self-organising emergent behaviour using advanced numerical methods*, Autonomic Computing, 2005. ICAC 2005. Proceedings. Second International Conference on, IEEE, 2005, pp. 52–63. 3.3.1

[EE90] Inc Electronics Engingeers, *Ieee std. 610.12-1990 - ieee standard glossary of software engineering terminology*, Office **121990** (1990), 84. 2.5.1, 7.4

[EH07] Clement Escoffier and Richard S. Hall, *Dynamically adaptable applications with ipojo service components*, Proceedings of the 6th international conference on Software composition (Berlin, Heidelberg), SC'07, Springer-Verlag, 2007, pp. 113–128. 3.1.2, 6, 6.1.1.1

[EHL07] C. Escoffier, R.S. Hall, and P. Lalanda, *ipojo: an extensible service-oriented component framework*, Services Computing, 2007. SCC 2007. IEEE International Conference on, july 2007, pp. 474 –481. 3.1.2, 6, 6.1.1.1

[emf13] *Eclipse modeling framework*, `http://www.eclipse.org/modeling/emf/`, 2013. 5.2, 5.2.1

[Esc08] Clement Escoffier, *ipojo : Un modele à composant a service flexible pour les systemes dynamiques*, Ph.D. thesis, UNIVERSITE JOSEPH FOURIER, 2008. 2.5.1

[FBGA02] Stephanie Forrest, Justin Balthrop, Matthew Glickman, and David Ackley, *Computation in the wild*, 2002. 3.3

[FC09] Jorge Fox and Siobhán Clarke, *Exploring approaches to dynamic adaptation*, MAI '09: Proceedings of the 3rd International DiscCoTec Workshop on Middleware-Application Interaction (New York, NY, USA), ACM, 2009, pp. 19–24. 2.5.1

[FDMD13] Sylvain Frey, Ada Diaconescu, David Menga, and Isabelle Demeure, *Towards a reference model for multi-goal, highly-distributed and dynamic autonomic systems*, International Conference on Autonomic Computing ICAC (San Jose, Ca, USA,), vol. Self-aware Internet of Things (Self-IoT) track, Juin 2013 2013. 2.5.1

[FFMM94] T. Finin, R. Fritzson, D. McKay, and R. McEntire, *Kqml as an agent communication language*, Proceedings of the third international conference on Information and knowledge management, ACM, 1994, pp. 456–463. 3.2.2

[FGG+06] Peter Feiler, Richard P Gabriel, John Goodenough, Rick Linger, Tom Longstaff, Rick Kazman, Mark Klein, Linda Northrop, Douglas Schmidt, Kevin Sullivan, et al., *Ultra-large-scale systems: The software challenge of the future*, Software Engineering Institute (2006). 1.1

[FIP01] FIPA, *Fipa dutch auction interaction protocol specification*, `http://www.fipa.org/specs/fipa00032/`, 02 2001. 3.2.2, 8.3.2

[FIP02] _____, *Agent communication language specifications*, 2002. 3.2.2

[Gar95] David Garlan, *Software architecture (panel): next steps towards an engineering discipline for software systems design*, SIGSOFT Softw. Eng. Notes **20** (1995), no. 4, 5–. 3.4

[Gar00] _____, *Software architecture: a roadmap*, Proceedings of the Conference on The Future of Software Engineering (New York, NY, USA), ICSE '00, ACM, 2000, pp. 91–101. 3.4

[GCH+04] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste, *Rainbow: Architecture-based self-adaptation with reusable infrastructure*, Computer **37** (2004), 46–54. 2.5.2

[GF92] M. Genesereth and R. Fikes, *Knowledge interchange format, version 3.0 reference*

manual, Tech. report, 1992. 3.2.1

[GF00] Olivier Gutknecht and Jacques Ferber, *Madkit: A generic multi-agent platform*, Proceedings of the fourth international conference on Autonomous agents, ACM, 2000, pp. 78–79. 3.2.5

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*, vol. 206, Addison-wesley Reading, MA, 1995. 6.3.2

[GHT09] John C. Georgas, André van der Hoek, and Richard N. Taylor, *Using architectural models to manage and visualize runtime adaptation*, Computer **42** (2009), no. 10, 52–60. 3.4.3, 5.1.2

[GIO09] T. Gjerlufsen, M. Ingstrup, and J.W. Olsen, *Mirrors of meaning: Supporting inspectable runtime models*, Computer **42** (2009), no. 10, 61–68. 3.4.3, 5.1.2

[GK94] Michael R. Genesereth and Steven P. Ketchpel, *Software agents*, Communications of the ACM **37** (1994), 48–53. 3.2.1

[GKLZ09] Kutila Gunasekera, Shonali Krishnaswamy, Seng Wai Loke, and Arkady Zaslavsky, *Runtime efficiency of adaptive mobile software agents in pervasive computing environments*, Proceedings of the 2009 international conference on Pervasive services (New York, NY, USA), ICPS '09, ACM, 2009, pp. 123–132. 3.2.3

[GMD+11] Issac Garcia, Denis Morand, Bassem Debbabi, Philippe Lalanda, and Pierre Bourret, *A reflective framework for mediation applications*, Adaptive and Reflective Middleware on Proceedings of the International Workshop (New York, NY, USA), ARM '11, ACM, 2011, pp. 22–28. 1.4, 4.2.2, 4.3.2, 6.4.4, 7.1.2, 7.1.2

[GMW97] David Garlan, Robert Monroe, and David Wile, *Acme: an architecture description interchange language*, Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research, CASCON '97, IBM Press, 1997, pp. 7–. 3.4.3

[GPD+10] I. Garcia, G. Pedraza, Bassem Debbabi, Philippe Lalanda, and C. Hamon, *Towards a service mediation framework for dynamic applications*, Services Computing Conference (APSCC), 2010 IEEE Asia-Pacific, 2010, pp. 3–10. 1.4, 7.1.2

[GR91] M.M. Gorlick and R.R. Razouk, *Using weaves for software construction and analysis*, Software Engineering, 1991. Proceedings., 13th International Conference on, IEEE, 1991, pp. 23–34. 3.4.3

[GS93] David Garlan and Mary Shaw, *An introduction to software architecture*, Advances in Software Engineering and Knowledge Engineering **1** (1993). 3.4

[GZKL08] K. Gunasekera, A. Zaslavsky, S. Krishnaswamy, and S.W. Loke, *Versag: Context-aware adaptive mobile agents for the semantic web*, Computer Software and Applications, 2008. COMPSAC '08. 32nd Annual IEEE International, 28 2008-aug. 1 2008, pp. 521 –522. 3.2.3

[HDBVDHCM99] Koen V. Hindriks, Frank S. De Boer, Wiebe Van Der Hoek, and John-Jules Ch. Meyer, *Agent programming in 3apl*, Autonomous Agents and Multi-Agent Systems **2** (1999), no. 4, 357–401. 3.2.5

[Hel09] Joseph L. Hellerstein, *Engineering autonomic systems*, ICAC, 2009, pp. 75–76. 3.1.2, 4.1

[HHPS08] S. Hallsteinsen, M. Hinchey, Sooyong Park, and K. Schmid, *Dynamic software product lines*, Computer **41** (2008), no. 4, 93–95. 3.4.3

[HIM+98] Dan Hirsch, Paola Inverardi, Ugo Montanari, Dep De Computacion, and Dipartamento Di Matematica, *Graph grammars and constraint solving for software architecture styles*, In Proc. of the Int. Software Architecture Workshop, ACM Press, 1998, pp. 69–72. 5.2.1

[HKJR10] Patrick Hunt, Mahadev Konar, Flavio P Junqueira, and Benjamin Reed, *Zookeeper: wait-free coordination for internet-scale systems*, Proceedings of the 2010 USENIX conference on USENIX annual technical conference, vol. 8, 2010, pp. 11–11. 8.3.2

[HL04] Bryan Horling and Victor Lesser, *A survey of multi-agent organizational paradigms*, Knowl. Eng. Rev. **19** (2004), 281–316. 3.2.1, 4.4.1

[HLM+09] Fabien Hermenier, Xavier Lorca, Jean-Marc Menaud, Gilles Muller, and Julia Lawall, *Entropy: a consolidation manager for clusters*, Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments (New York, NY, USA), VEE '09, ACM, 2009, pp. 41–50. 3.1.2

[HM08] Markus C. Huebscher and Julie A. McCann, *A survey of autonomic computing—degrees, models, and applications*, ACM Comput. Surv. **40** (2008), no. 3, 1–28. 2.1.2, 2.5.2

[HMNS03] Uwe Hansmann, Lothar Merk, Martin S. Nicklous, and Thomas Stober, *Pervasive computing : The mobile world*, 2nd ed., Springer, August 2003. 1.1

[Hor01] Horn, *Autonomic computing: Ibm's perspective on the state of information technology*, IBM Corporation (October 15, 2001). Available at http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf (2001). 1.1, 2.1.1, 2.1.2, 2.2, 3.3

[HRHL01] N. Howden, R. Ronnquist, A. Hodgson, and A. Lucas, *Jack intelligent agents - summary of an agent infrastructure.*, Proceedings of the 5th ACM International Conference on Autonomous Agents, 2001. 3.2.5

[HS07] M.G. Hinchey and R. Sterritt, *99% (biological) inspiration...* 3.3

[HTL05] C. Herault, G. Thomas, and P. Lalanda, *Mediation and enterprise service bus a position paper*, Proceedings of the First International Workshop on Mediation in Semantic Web Services (MEDIATE 2005), Citeseer, 2005. 7.1.1

[HTL07] _____, *A distributed service-oriented mediation tool*, IEEE International Conference on Services Computing, 2007. SCC 2007, 2007, pp. 403–409. 7.1.1

[HW05] Aaron Helsinger and Todd Wright, *Cougaar: A robust configurable multi agent platform*, Aerospace Conference, 2005 IEEE, IEEE, 2005, pp. 1–10. 2.4.3, 3.2.5

[IBM03] IBM, *An architectural blueprint for autonomic computing*, IBM White Paper (2003). 2.4.1

[Ins13] Software Engineering Institue, *Software product lines*, `http://www.sei.cmu.edu/productlines/`, 2013, [Online; accessed 08-March-2013]. 3.4.3

[Jen95] N. R. Jennings, *Controlling cooperative problem solving in industrial multi-agent systems using joint intentions*, Artif. Intell. **75** (1995), no. 2, 195–240. 3.2.2

[JFL+01] N. R. Jennings, P. Faratin, A. R. Lomuscio, S. Parsons, C. Sierra, and M. Wooldridge,

Automated negotiation: Prospects, methods and challenges, 2001. 3.2.2

[JLHNY04] Bart Jacob, Richard Lanyon-Hogg, Devaprasad K Nadgir, and Amr F Yassin, *A practical guide to the ibm autonomic computing toolkit*, 2004. 2.5.2

[JQSP08] Nanyan Jiang, Andres Quiroz, Cristina Schmidt, and Manish Parashar, *Meteor: a middleware infrastructure for content-based decoupled interactions in pervasive grid environments*, Concurr. Comput. : Pract. Exper. **20** (2008), no. 12, 1455–1484. 2.5.2

[JSW98] Nicholas R. Jennings, Katia Sycara, and Michael Wooldridge, *A roadmap of agent research and development*, Autonomous Agents and Multi-Agent Systems **1** (1998), no. 1, 7–38. 3.2, 3.2.1, 3.2.3

[JW00] Nicholas R. Jennings and Michael Wooldridge, *Agent-oriented software engineering*, ARTIFICIAL INTELLIGENCE **117** (2000), 277–296. 3.2.3

[KBE99] Mieczyslaw M. Kokar, Kenneth Baclawski, and Yonet A. Eracar, *Control theory-based foundations of self-controlling software*, IEEE Intelligent Systems **14** (1999), no. 3, 37–45. 3.1, 3.1.2

[KC03] Jeffrey O. Kephart and David M. Chess, *The vision of autonomic computing*, Computer **36** (2003), no. 1, 41–50. 2.1.1, 2.1.2, 2.3, 2.4.1, 3.2.4, 5.4.1

[KKH+09] Dara Kusic, Jeffrey O. Kephart, James E. Hanson, Nagarajan Kandasamy, and Guofei Jiang, *Power and performance management of virtualized computing environments via lookahead control*, Cluster Computing **12** (2009), no. 1, 1–15. 3.1.2

[KLD02] Kyo C. Kang, Jaejoon Lee, and Patrick Donohoe, *Feature-oriented product line engineering*, IEEE Software **19** (2002), no. 4, 58–65. 3.4.3

[KW04] J.O. Kephart and W.E. Walsh, *An artificial intelligence perspective on autonomic computing policies*, Policies for Distributed Systems and Networks, 2004. POLICY 2004. Proceedings. Fifth IEEE International Workshop on, june 2004, pp. 3 – 12. 2.4.4

[KWB03] Anneke G. Kleppe, Jos Warmer, and Wim Bast, *Mda explained: The model driven architecture: Practice and promise*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. 5.1.2

[LBCP09] Harold C. Lim, Shivnath Babu, Jeffrey S. Chase, and Sujay S. Parekh, *Automated control in cloud computing: challenges and opportunities*, Proceedings of the 1st workshop on Automated control for datacenters and clouds (New York, NY, USA), ACDC '09, ACM, 2009, pp. 13–18. 3.1.2

[LDJ13] Philippe Lalanda, Ada Diaconescu, and A. Julie, Mccann, *Autonomic computing - principles, design and implementation*, Undergraduate Topics in Computer Science, Springer, May 2013 (Anglais). (document), 2.1.2, 2.4.1, 2.1, 2.4.2, 2.4.2, 2.4.3

[LKA+95] David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann, *Specification and analysis of system architecture using rapide*, IEEE Transactions on Software Engineering **21** (1995), 336–355. 3.4.3

[LML05] P. Lin, A. MacArthur, and J. Leaney, *Defining autonomic computing: a software engineering perspective*, Proc. Australian Software Engineering Conf, 2005, pp. 88–97. 2.1.2, 2.3, 2.5.1

[Lod04] K.N. Lodding, *The hitchhiker's guide to biomorphic software*, Queue **2** (2004), no. 4, 66–75. 3.3

[LP04] Zhen Li and M. Parashar, *Rudder: a rule-based multi-agent infrastructure for supporting autonomic grid applications*, Autonomic Computing, 2004. Proceedings. International Conference on, may 2004, pp. 278 – 279. 2.5.2

[LPH04] Hua Liu, M. Parashar, and S. Hariri, *A component-based programming model for autonomic applications*, Autonomic Computing, 2004. Proceedings. International Conference on, may 2004, pp. 10 – 17. 2.5.2

[LPM06] Hua Liu, Manish Parashar, and Senior Member, *Accord: A programming framework for autonomic applications*, IEEE Transactions on Systems, Man and Cybernetics, Special Issue on Engineering Autonomic Systems, Editors: R. Sterritt and T. Bapty, IEEE Press **36** (2006), 341–352. 2.5.2

[LRS97] Robert Laddaga, Paul Robertson, and Howard E Shrobe, *Self-adaptive software*, Proposer Information Pamphlet BAA (1997), no. 98-12. 3.1.2

[LS99] Ora Lassila and Ralph R. Swick, *Resource description framework (rdf) model and syntax specification*, Recommendation 22 February 1999 REC-rdf-syntax-19990222, W3C, Cambridge, MA, 1999. 5.3

[Mai02] E. Mainsah, *Autonomic computing: the next era of computing*, Electronics Communication Engineering Journal **14** (2002), no. 1, 2 –3. 2.1.1, 3.3

[Mao09] Shahar Maoz, *Using model-based traces as runtime models*, Computer **42** (2009), no. 10, 28–36. 3.4.3, 5.1.2

[Mar00] P. Marrow, *Nature-inspired computing technology and applications*, BT Technology Journal **18** (2000), no. 4, 13–23. 3.3

[Mau10] Yoann Maurel, *Ceylon: Un canevas pour la creation de gestionnaires autonomiques extensibles et dynamiques*, Ph.D. thesis, University of Grenoble, 2010. 2.5.2

[MBJ+09] Brice Morin, Olivier Barais, Jean-Marc Jezequel, Franck Fleurey, and Arnor Solberg, *Models@ run.time to support dynamic adaptation*, Computer **42** (2009), no. 10, 44–51. 3.4.3, 5.1.2

[MCC04] Matthew L Massie, Brent N Chun, and David E Culler, *The ganglia distributed monitoring system: design, implementation, and experience*, Parallel Computing **30** (2004), no. 7, 817–840. 2.4.3

[MDL10] Yoann Maurel, Ada Diaconescu, and Philippe Lalanda, *Ceylon: A service-oriented framework for building autonomic managers*, Proceedings of the 2010 Seventh IEEE International Conference and Workshops on Engineering of Autonomic and Autonomous Systems (Washington, DC, USA), EASE '10, IEEE Computer Society, 2010, pp. 3–11. 2.5.2

[MFB+08] Brice Morin, Franck Fleurey, Nelly Bencomo, Jean-Marc Jézéquel, Arnor Solberg, Vegard Dehlen, and Gordon Blair, *An aspect-oriented and model-driven approach for managing dynamic variability*, Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems (Berlin, Heidelberg), MoDELS '08, Springer-Verlag, 2008, pp. 782–796. 3.4.3

[MGGL11] Denis Morand, Isaac Noé Garcia Garza, and Philippe Lalanda, *Autonomic enterprise service bus*, Proceedings of the 2011 IEEE 16th Conference on Emerging Technologies & Factory Automation (ETFA) (Toulouse, France), IEEE Conference Publications, September 2011, pp. 1–8 (Anglais). 2.4.3

[Mil05] Brent Miller, *The autonomic computing edge: The "standard" way of autonomic computing*, March 2005. 2.5.2

[MK96] Jeff Magee and Jeff Kramer, *Dynamic structure in software architectures*, SIGSOFT Softw. Eng. Notes **21** (1996), no. 6, 3–14. 3.4.3

[MMWW06] Hausi Muller, Klein Mark, Wood William, and O'Brien William, *Autonomic computing*, Tech. Report CMU/SEI-2006-TN-006, Software Engineering Institute, Carnegie Mellon University, 2006. 2.1.2

[MSKC04] P.K. McKinley, S.M. Sadjadi, E.P. Kasten, and B.H.C. Cheng, *Composing adaptive software*, Computer **37** (2004), no. 7, 56–64. 2.1.2, 3.1

[MT89] Mihajlo D. Mesarovic and Yasuhiko Takahara, *Abstract systems theory*, Lecture notes in control and information sciences, Springer-Verlag, Berlin, New York, 1989. 3.1

[MT00] Nenad Medvidovic and Richard N. Taylor, *A classification and comparison framework for software architecture description languages*, IEEE Trans. Softw. Eng. **26** (2000), no. 1, 70–93. 3.4.3

[Nag04] Radhika Nagpal, *A catalog of biologically-inspired primitives for engineering self-organization.*, Engineering Self-Organising Systems, Nature-Inspired Approaches to Software Engineering., Lecture Notes in Computer Science, vol. 2977, Springer, 2004, pp. 53–62. 3.3

[OMG13] OMG, *Model driven architecture*, `http://www.omg.org/mda`, 2013. 5.1.2

[OMT08] P. Oreizy, N. Medvidovic, and R.N. Taylor, *Runtime software adaptation: framework, approaches, and styles*, Companion of the 30th international conference on Software engineering, ACM, 2008, pp. 899–910. 3.4.2

[Oqu04] Flavio Oquendo, *Pi-adl: an architecture description language based on the higher-order typed pi-calculus for specifying dynamic and mobile software architectures*, SIGSOFT Softw. Eng. Notes **29** (2004), no. 3, 1–14. 3.4.3

[Ore98] *Architecture-based runtime software evolution*, 1998. 3.4, 3.4.2

[Ore00] Peyman Oreizy, *Open architecture software: a flexible approach to decentralized software evolution*, Ph.D. thesis, 2000, AAI9963052. 3.4.3

[OS06] Reza. Olfati-Saber, *Flocking for multi-agent dynamic systems: algorithms and theory*, Automatic Control, IEEE Transactions on **51** (2006), no. 3, 401 – 420. 3.3.1

[OVDPB01] James Odell, H Van Dyke Parunak, and Bernhard Bauer, *Representing agent interaction protocols in uml*, Agent-Oriented Software Engineering, Springer, 2001, pp. 201–218. 3.2.5

[OVDPFB03] James J. Odell, H. Van Dyke Parunak, Mitch Fleischer, and Sven Brueckner, *Modeling agents and their environment*, Proceedings of the 3rd international conference on Agent-oriented software engineering III (Berlin, Heidelberg), AOSE'02, Springer-Verlag, 2003, pp. 16–31. 3.2, 3.2.2

[PCHW12] T. Patikirikorala, A. Colman, Jun Han, and Liuping Wang, *A systematic survey on the design of self-adaptive software systems using control engineering approaches*, Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2012 ICSE Workshop on, june 2012, pp. 33 –42. 3.1.1, 3.1.2

[PCT03] David Pierre Charles and Ledoux Thomas, *Towards a framework for self-adaptive*

component-based applications, In DAIS'03, volume 2893 of LNCS, Springer-Verlag, 2003, pp. 1–14. 3.1.2

[PH06] M Parashar and S Hariri, *Autonomic grid computing concepts, requirements, infrastructures*, 2006. 2.1.2

[PLL+06] Manish Parashar, Hua Liu, Zhen Li, Vincent Matossian, Cristina Schmidt, Guangsen Zhang, and Salim Hariri, *Automate: Enabling autonomic applications on the grid*, Cluster Computing **9** (2006), no. 2, 161–174. (document), 2.5.2, 2.8, 4.1

[PW92] Dewayne E. Perry and Alexander L. Wolf, *Foundations for the study of software architecture*, SIGSOFT Softw. Eng. Notes **17** (1992), no. 4, 40–52. 3.4

[PWB09] Evangelos Pournaras, Martijn Warnier, and Frances M. T. Brazier, *Adaptive agent-based self-organization for robust hierarchical topologies*, Proceedings of the 2009 International Conference on Adaptive and Intelligent Systems (Washington, DC, USA), ICAIS '09, IEEE Computer Society, 2009, pp. 69–76. 3.2.3

[RCL09] Bhaskar Prasad Rimal, Eunmi Choi, and Ian Lumb, *A taxonomy and survey of cloud computing systems*, Proceedings of the 2009 Fifth International Joint Conference on INC, IMS and IDC (Washington, DC, USA), NCM '09, IEEE Computer Society, 2009, pp. 44–51. 1.1

[RvBW06] Francesca Rossi, Peter van Beek, and Toby Walsh, *Handbook of constraint programming (foundations of artificial intelligence)*, Elsevier Science Inc., New York, NY, USA, 2006. 5.4.3

[Sam97] Johannes Sametinger, *Software engineering with reusable components*, Springer-Verlag New York, Inc., New York, NY, USA, 1997. 2.5.1

[SBDP08] Sylvain Sicard, Fabienne Boyer, and Noel De Palma, *Using components for architecture-based management: the self-repair case*, Proceedings of the 30th international conference on Software engineering (New York, NY, USA), ICSE '08, ACM, 2008, pp. 101–110. 2.5.2

[Sch93] Alexander Schill (ed.), *Proceedings of the international dce workshop on dce - the osf distributed computing environment, client/server model and beyond*, London, UK, UK, Springer-Verlag, 1993. 5.2.4

[Sed04] Igor Sedukhin, *Web services distributed management: Management of web services (wsdm-mows) 1.0*, OASIS OASIS Web Services Distributed Management (WSDM) TC (2004). 2.5.2

[SG96] Mary Shaw and David Garlan, *Software architecture: Perspectives on an emerging discipline*, Prentice Hall, April 1996, http://www.docin.com/p-41442459.html. 3.4, 3.4.1

[SH05] Roy Sterritt and Mike Hinchey, *Autonomic computing-panacea or poppycock?*, Engineering of Computer-Based Systems, 2005. ECBS'05. 12th IEEE International Conference and Workshops on the, IEEE, 2005, pp. 535–539. 2.3

[SH09] _____, *Safety and security in multiagent systems*, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 330–341. 3.3.3

[SHC+11] Hui Song, Gang Huang, Franck Chauvel, Yingfei Xiong, Zhenjiang Hu, Yanchun Sun, and Hong Mei, *Supporting runtime software architecture: A bidirectional-*

transformation-based approach, J. Syst. Softw. **84** (2011), no. 5, 711–723. 4.4.4

[SKPF03] Lee Spector, Jon Klein, Chris Perry, and Mark Feinstein, *Emergence of collective behavior in evolving populations of flying agents*, Proceedings of the 2003 international conference on Genetic and evolutionary computation: PartI (Berlin, Heidelberg), GECCO'03, Springer-Verlag, 2003, pp. 61–73. 3.3.1

[Smi80] R.G. Smith, *The contract net protocol: High-level communication and control in a distributed problem solver*, Computers, IEEE Transactions on **100** (1980), no. 12, 1104–1113. 3.2.2, 8.3.2

[SP08] Candelaria Sansores and Juan Pavón, *An adaptive agent model for self-organizing mas*, Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems - Volume 3 (Richland, SC), AAMAS '08, International Foundation for Autonomous Agents and Multiagent Systems, 2008, pp. 1639–1642. 3.2.3

[SPTU05] Roy Sterritt, Manish Parashar, Huaglory Tianfield, and Rainer Unland, *A concise introduction to autonomic computing*, Advanced Engineering Informatics **19** (2005), no. 3, 181–187. 2.1.2

[ST09] M. Salehie and L. Tahvildari, *Self-adaptive software: Landscape and research challenges*, ACM Transactions on Autonomous and Adaptive Systems (TAAS) **4** (2009), no. 2, 1–42. 2.1.2, 3.1.2

[Ste05] Roy Sterritt, *Autonomic computing*, Innovations in systems and software engineering **1** (2005), no. 1, 79–88. 2.1.2

[Sun11] E.C. Sun, *Ant colonies: Behavior in insects and computer applications*, Computer Science, Technology and Applications, Nova Science Pub Incorporated, 2011. 3.3.1

[Tal12] Domenico Talia, *Clouds meet agents: Toward intelligent cloud services*, IEEE Internet Computing **16** (2012), no. 2, 78–81. 3.2.4

[TCW+04] Gerald Tesauro, David M. Chess, William E. Walsh, Rajarshi Das, Alla Segal, Ian Whalley, Jeffrey O. Kephart, and Steve R. White, *A multi-agent systems approach to autonomic computing*, Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 1 (Washington, DC, USA), AAMAS '04, IEEE Computer Society, 2004, pp. 464–471. 2.5.2, 3.2.4

[TDBH09] Christophe Taton, Noël De Palma, Sara Bouchenak, and Daniel Hagimont, *Improving the performance of jms-based applications*, International Journal of Autonomic Computing (IJAC) **1** (2009), no. 1. 3.1.2

[TMO09] R.N. Taylor, N. Medvidovic, and P. Oreizy, *Architectural styles for runtime software adaptation*, WICSA/ECSA **9** (2009). 3.4.3

[UD11] Mihaela Ulieru and Rene Doursat, *Emergent engineering: a radical paradigm shift*, Int. J. Auton. Adapt. Commun. Syst. **4** (2011), no. 1, 39–60. 3.3.1

[wCcHG+04] Shang wen Cheng, An cheng Huang, David Garlan, Bradley Schmerl, and Peter Steenkiste, *An architecture for coordinating multiple self-management systems*, IEEE/IFIP Conference on Software Architecture, Kluwer Academic Publishers, 2004, pp. 12–15. 2.5.2

[Wei99] G. Weiss, *Multiagent systems: a modern approach to distributed artificial intelligence*, MIT press, 1999. (document), 3.2, 3.7

[WG97] G. Wiederhold and M. Genesereth, *The conceptual basis for mediation services*, IEEE Expert, Vol. 12 No. 5, Sep.-Oct. 1997, pages 38-47 (1997). 1.4, 7.1.1, 7.1.1

[WH05] Tom De Wolf and Tom Holvoet, *Emergence versus self-organisation: Different concepts but promising when combined*, Springer-Verlag, 2005, pp. 1–15. 3.3.1

[WH06] M. Wirsing and M. Holzl, *Software intensive systems*, Draft Report on ERCIM Beyond the Horizon Thematic Group **6** (2006). (document), 1.1

[Wie92] G. Wiederhold, *Mediators in the architecture of future information systems*, Computer **25** (1992), no. 3, 38–49. 1.4, 3.2.1, 7.1.1

[WJK00] Michael Wooldridge, Nicholas R Jennings, and David Kinny, *The gaia methodology for agent-oriented analysis and design*, Autonomous Agents and Multi-Agent Systems **3** (2000), no. 3, 285–312. 3.2.5

[Woo97] M. Wooldridge, *Agent-based software engineering*, Software Engineering. IEE Proceedings- [see also Software, IEE Proceedings] **144** (1997), no. 1, 26 –37. 3.2

[YGS+04] Hong Yan, David Garlan, Bradley Schmerl, Jonathan Aldrich, and Rick Kazman, *Discotect: A system for discovering architectures from running systems*, Proceedings of the 26th International Conference on Software Engineering (Washington, DC, USA), ICSE '04, IEEE Computer Society, 2004, pp. 470–479. 2.5.2

[YKO03] O. Yadgar, S. Kraus, and C.L. Ortiz, *Scaling-up distributed sensor networks: cooperative large-scale mobile-agent organizations*, Distributed Sensor Networks (2003), 185–217. 3.2.1, 3.2.1

[ZC06] Ji Zhang and Betty H. C. Cheng, *Model-based development of dynamically adaptive software*, ICSE '06: Proceedings of the 28th international conference on Software engineering (New York, NY, USA), ACM, 2006, pp. 371–380. 3.4.3, 5.1.2

[ZJW03] Franco Zambonelli, Nicholas R. Jennings, and Michael Wooldridge, *Developing multiagent systems: The gaia methodology*, ACM Trans. Softw. Eng. Methodol. **12** (2003), no. 3, 317–370. 3.2.3