Building manageable autonomic control loops for large scale systems

Russel Nzekwa

To cite this version:

Russel Nzekwa. Building manageable autonomic control loops for large scale systems. Software Engineering [cs.SE]. Université des Sciences et Technologie de Lille - Lille I, 2013. English. NNT : . tel-00843874

HAL Id: tel-00843874
https://theses.hal.science/tel-00843874

Submitted on 12 Jul 2013

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.

Département de formation doctorale en informatique
UFR IEEA

École doctorale SPI Lille

Building Manageable Autonomic Control Loops for Large Scale Systems

THÈSE

présentée et soutenue publiquement le 05/7/2013

pour l'obtention du

Doctorat de l'université Lille 1 Sciences et Technologies

(spécialité informatique)

par

Russel Nzekwa

Composition du jury

Rapporteurs :	Jean-charles Fabre - Professeur des Universités, *ENSEEIHT, Toulouse- France*
	Sara Bouchenak - Maître de Conférences, *Université de Grenoble - France*
Examinateurs :	Jacques Malenfant - Professeur des Universités, *Université Pierre et Marie Curie - France*
	Eric Rutten - Chargé de Recherche, *LIG/INRIA Grenoble - France*
Directeurs :	Lionel Seinturier - Professeur des Universités, *Université Lille I et IUF - France*
	Romain Rouvoy - Maître de Conférences, *Université Lille I - France*

LIFL – UMR Université Lille 1 Sciences et Technologies/CNRS 8022 – INRIA Lille Nord Europe

Mis en page avec la classe thloria.

A mes parents, Jacob et Paulette Nana
pour leur soutient et leur amour inconditionnel

Remerciements

Cette thèse fut un parcours à la fois intellectuel et personnel. J'aimerai par conséquent saisir l'opportunité de ce moment qui sanctionne la fin de cette entreprise pour remercier tous ceux qui de près ou de loin ont contribué à rendre possible ce travail même s'ils ne sont pas nommément cités dans la suite.

En premier lieu, j'aimerai remercier mes encadrants Lionel et Romain. Lionel pour sa disponibilité et ses conseils bienveillants durant ces années. Romain, pour ses idées et son investissement constant. J'aimerai en deuxième lieu remercier les membres du jury Sara Bouchenak et Jean-Charles Fabre pour avoir accepté de rapporter ma thèse. J'adresse mes remerciements à Eric Rutten pour avoir accepté de participer au jury de ma thèse et à Jacques Malenfant pour avoir accepté de présider ce jury.

Je voudrai adresser mes vifs remerciements à tous les membres de l'équipe INRIA ADAM. En particulier à Laurence Duchien qui a toujours promu un esprit de fraternité au sein de l'équipe, et à Philippe Merle pour ses conseils sur les aspects du développement avec FraSCAti. Je tiens aussi à remercier mon collègue Daniel Romero avec qui nous avons beaucoup discuté et fait du jogging durant ces années.

Je voudrai aussi remercier les membres du projet SALTY avec qui j'ai collaboré durant ces années. En particulier, Filip Krikava avec qui j'ai échangé sur de nombreux aspects techniques dans le cadre du projet. J'aimerai aussi remercier les membres de l'équipe associée de l'université d'Oslo en particulier Frank Eliassen, pour l'accueil et l'accompagnement. Merci également aux différentes assistantes de l'équipe qui n'ont pas ménagé d'efforts pour nous accompagner dans la gestion de nos fiches de mission.

Je ne terminerai pas sans remercier mes amis qui m'ont soutenu et parfois inspiré durant cette période. Je pense en particulier à Mathieu, Jean et Rémi. Enfin mais pas des moindres, je voudrai remercier ma famille, en premier mes parents, Jacob et Paulette pour leur soutient inébranlable, mais aussi mes frères Joel, Michael, Gauis, Franklin et ma soeur Claudia pour le réconfort qu'ils m'ont apporté.

Abstract

Keywords: Autonomic systems, Feedback Control Loops, Stabilization, Service, Adaptation, Component, Model driven engineering

Modern software systems are getting more complex. This is partly justified by the heterogeneity of technologies embedded to deliver services to the end client, the large-scale distribution of software pieces that intervene within a single application, or the requirements for adaptive software systems. In addition, the need for reducing the maintenance costs of software systems has led to the emergence of new paradigms to cope with the complexity of these software. Autonomic computing is a relatively new paradigm for building software systems which aims at reducing the maintenance cost of software by building autonomic software systems which are able to manage themselves with a minimal intervention of a human operator.

However, building autonomic software raises many scientific and technological challenges. For example, the lack of visibility of the control system architecture in autonomic systems makes them difficult to maintain. Similarly, the lack of verification tools is a limitation for building reliable autonomic systems. The flexible management of non-functional-properties, or the traceability between the design and the implementation are other challenges that need to be addressed for building flexible autonomic systems.

The main contribution of this thesis is *CORONA*. *CORONA* is a framework which aims at helping software engineers for building flexible autonomic systems. To achieve that goal, *CORONA* relies on an architectural description language which reifies the structure of the control system architecture. *CORONA* enables the flexible integration of non-functional-properties during the design of autonomic systems. It also provides tools for conflicts checking in autonomic systems architecture. Finally, the traceability between the design and the runtime implementation is carried out through the code generation of skeletons from architectural descriptions of control systems. These properties of the *CORONA* framework are exemplified through three case-studies.

Résumé

Mots-clés: Applications autonomes, Boucle de contrôle rétroactive, Stabilisation, Service, Adaptation, Composant, Ingénierie dirigé par les modéles.

Les logiciels modernes sont de plus en plus complexes. Ceci est dû en partie à l'hétérogénéité des solutions misent en oeuvre, au caractère distribué des architectures de déploiement et à la dynamicité requise pour de tels logiciels qui devraient être capable de s'adapter en fonction des variations de leur contexte d'évolution. D'un autre coté, l'importance grandissante des contraintes de productivité dans le but de réduire les coûts de maintenance et de production des systèmes informatiques a favorisé l'émergence de nouveaux paradigmes pour répondre à la complexité des logiciels modernes. L'informatique des systèmes autonomes (Autonomic computing) s'inscrit dans cette perspective. Elle se propose entre autres de réduire le coût de maintenance des systèmes informatiques en développant des logiciels dits autonomes, c'est à dire dotés de la capacité de s'auto-gérer moyennant une intervention limité d'un opérateur humain.

Toutefois, le développement de logiciels autonomes soulèvent de nombreux défis scientifiques et technologiques. Par exemple, l'absence de visibilité de la couche de contrôle dans les applications autonomes rend difficile leur maintenabilité, l'absence d'outils de vérification pour les architectures autonomes est un frein pour l'implémentation d'applications fiables, enfin, la gestion transparente des propriétés-non-fonctionnelles et la traçabilité entre le design et l'implémentation sont autant de défis que pose la construction de logiciels autonomes flexibles.

La principale contribution de cette thèse est *CORONA*. *CORONA* est un canevas logiciel qui vise à faciliter le développement de logiciels autonomes flexibles. Dans cet objectif, *CORONA* s'appuie sur un langage de description architecturale qui réifie les éléments qui forment la couche de contrôle dans les systèmes autonomes. *CORONA* permet l'intégration transparente des propriétés-non-fonctionnelles dans la description architecturale des systèmes autonomes. il fournit aussi dans sa chaîne de compilation un ensemble d'outils qui permet d'effectuer des vérifications sur l'architecture des systèmes autonomes. Enfin, la traçabilité entre le design et l'implémentation est assurée par un mécanisme de génération des skeletons d'implémentation à partir d'une description architecturale. Les différentes propriétés de *CORONA* sont illustrées par trois cas d'utilisation.

Contents

List of Tables xv

Chapter 1 Introduction 1

1.1 Introduction 1

1.2 Problem Statement 3

1.3 Dissertation Goals 4

1.4 Contributions 5

1.5 Dissertation Roadmap 6

1.6 Publications 7

Part I State of the Art 9

Chapter 2 State of the Art 11

2.1 Autonomic Computing 12

2.2 Existing Autonomic Systems Approaches 16

2.3 Assessing Autonomic Systems 26

2.4 Summary 30

Chapter 3 Salty Model 33

3.1 SALTY Structural Model 34

3.2 SALTY Graphical Formalism 37

3.3 SALTY DSL 39

3.4 Summary 42

Part II Contribution 43

Chapter 4 Contributions Overview 45

4.1 Challenges Revisited 46

4.2 Goals Revisited 47

4.3 CORONA in a Nutshell 48

4.4 Summary 51

Chapter 5 Runtime Architecture 53

5.1 Feedback Control Loops and Autonomic Systems 54

5.2 Runtime Component-based Feedback Control Loops 56

5.3 Feedback Control Loop Customization 67

5.4 Summary 79

Chapter 6 Compilation Infrastructure 81

6.1 Component-based Generative ToolChain 82

6.2 Mapping from SALTY Model to SCA Model 85

6.3 Control Loop Architecture Distribution 89

6.4 Conflicts Checking on Feedback Control Loop Architectures 93

6.5 Control Loop Architecture Evolution 105

6.6 Summary 106

Part III Validation 109

Chapter 7 Condor Case-Study 111

7.1 Case-study Objective 112

7.2 Condor Case-Study Description 112

7.3 Control System Architecture 113

7.4 Quantitative Evaluation 115

7.5 Summary 120

Chapter 8 Fire Emergency Case-Study 123

8.1 Case-Study Objective 124

8.2 Scenario Description 124

8.3 Control System Architecture 125

8.4 Control System Implementation & Measures 126

8.5 Summary 130

Chapter 9 Smart-Mall Case-Study 131

9.1 Objective 131

9.2 Smart-Mall Scenario Description 132

9.3 Experiment & Measures 134

9.4 Summary 139

Part IV Conclusion & Persperctives 141

Chapter 10 Conclusion 143

10.1 Summary of the Dissertation 143

10.2 Perspectives 146

Bibliography 149

List of Figures

2.1 Structure of an Autonomic Element [Kep05] 14
2.2 Overview of an Autonomic System Architecture 15
2.3 Overview of the Jade Approach [BPHT06] 17
2.4 Architecture of Unity Autonomic System 19
2.5 Architecture of an Autonomic Agent in Autonomia 20
2.6 Rainbow System Architecture with Customization Points [wC08] 21
2.7 Ceylan Autonomic Control Architecture [MDL10] 23
2.8 Control Architectural Style in DiaSpec 24

3.1 SALTY Graphical Notations 38
3.2 Example of Apache Control Architecture Representation with the SALTY Specification 38
3.3 Annotation Class Diagram 41

4.1 Overview Of the *CORONA* Development Process 49
4.2 SCA Architecture of the *CORONA* Toolchain 50

5.1 Close And Open Loop Paradigms 55
5.2 Basic concepts of the SCA Metamodel 58
5.3 Graphical Representation of a Component in SCA 58
5.4 Auto-Scale Feedback Control Architecture 59
5.5 Generated Auto-Scale Feedback Control Loop Architecture in SCA 61

5.6 Generated Sensors Artifacts 62
5.7 Customizable MAPE-K Architecture Model 68
5.8 Feedback Block Diagram in Control Engineering 70
5.9 Unstable Auto-Scale Feedback Control System 70
5.10 Classification of Stabilization Algorithms According to their Class and Cost . 76
5.11 Relationships between algorithms of the first(a), second (b) and third (c) layer of the classification 77
5.12 Stabilization Algorithms Composition Models 78

6.1 ToolChain Key Features Behavior 83
6.2 SCA Architecture of the *CORONA* Compiler 83
6.3 Condor [TTL05] Control Loop Architecture 86
6.4 Illustration of Possible ambiguity in the interaction model 88
6.5 Basic Concepts of the Network Meta-Model 90
6.6 Illustration of a Feedback Control Loop Architecture with Constraints Annotations 92
6.7 Computation Result of the location Optimizer Service 93
6.8 Illustration of the Invisible Interference Problem 94
6.9 Conflicts verification Process 96
6.10 Direct overlaps Patterns 97
6.11 Transitive overlap Example 98
6.12 (A)– Control Architecture with Conflicts, (B)– Resolution of Architecture Conflicts with Proxy Pattern 102
6.13 (A)– Resolution of Conflicts with the Supervisor Mechanism, (B)– SCA implementation of the Supervisor Mechanism on an Effector 103
6.14 Supervisor Mechanism Coordination Logic 104
6.15 Control Loop Architecture Evolution Cycle 105
6.16 Control Architecture Selective Generation 106

7.1 Self-Adaptive Distributed Infrastructure 113
7.2 Self-Scale Feedback Loop 114
7.3 Budget Feedback Loop 114

7.4 Feedback Loops Overlaps Detection Time 116
7.5 System without Feedback Loops 118
7.6 System With a Single Budget Feedback Loop 118
7.7 System With a Single User SLA Feedback loop 119
7.8 System with Uncoordinated Feedback Loops 119
7.9 coordinated Control System Architecture 121
7.10 System with Coordinated Feedback Loops 121

8.1 Illustration of the Fire-emergency Scenario 125
8.2 Architecture of the fire-emergency Control System 126
8.3 Fire-emergency SCA assembly architecture 129

9.1 Smart-Mall Scenario illustration 133
9.2 Showroom Control System Architecture 135
9.3 Stabilized application behavior with Kalman Filter or Delta Operator 136
9.4 Stabilized application behavior with Horizontal Composition (DO+KF) . . . 136
9.5 Media Player Control System Architecture 137
9.6 Variation of Triggered Adaptations 138
9.7 Precision of Context Stabilization 138

List of Tables

2.1 Classification of Architecture-based Autonomic Solutions 29

5.1 Characterization of Stabilization Algorithms According to the Data Scope Criterion. 74

5.2 Analytic Definition of Algorithmic Process Behavior Classes 75

6.1 Mapping Rules Between SALTY and SCA Concepts 88

8.1 Metrics of generated and Implemented Code 129

This page was intentionally left blank

Chapter 1

Introduction

"Civilization advances by extending the number of important operations which we can perform without thinking about them."-Alfred N. Whitehead

Contents

1.1 Introduction 1
1.2 Problem Statement 3
1.3 Dissertation Goals 4
1.4 Contributions 5
1.5 Dissertation Roadmap 6
1.6 Publications 7

1.1 Introduction

Think in a wink about "*computing systems capable of running themselves, adjusting to varying circumstances, and preparing their resources to handle most efficiently the workloads we put upon them; capable to anticipate their needs and allow users to concentrate on what they want to accomplish rather than figuring how to rig the computing system to get them there*" [IBM01]. It is in this terms that Paul Horn, senior vice president of IBM research, presented in 2001 his vision of the future of information systems. This new vision of software systems is the subject of the autonomic computing.

Autonomic computing is a nearly recent research field that addresses the increasing complexity of software systems. Modern software applications are distributed, and are build from heterogeneous components. This make them very difficult to maintain. In particular, the growth of software complexity has blew up the cost of maintenance. Some estimate that, the number of IT workers required to support billions of users interacting with millions of

business applications connected via Internet is about 200 million, which is almost the population of a country like the United State of America. Recently in France, the 6th of July 2012, customers of a well-established telecommunication company have experienced a blackout due to a critical software failure. For several hours, customers have not been able to receive text messages or make phone calls. More than 200 engineers were mobilized to solve the failure, and for hours could not figure where the problem was. The company estimated to 30 million euros, the cost related to that failure. Undoubtedly, the actual complexity of software infrastructures is a threat that could lead information technology to a serious crisis.

Up until recent years, the maintenance cost of software systems was not a matter of concerns. Indeed, the improvement of hardware performance has been the main focus of the research and the industry community over these last decades. In particular, the fulfillment of predictions of Moore's law has allowed to develop faster and cheaper systems. In the same time, software development techniques were improving, and the emergence of object oriented languages in the early 90's, and components oriented languages a bit later, have significantly increased the quality of software systems. However, the advent of Internet has given rise to a new class of software systems that are *heterogeneous*, *large-scale*, and *distributed*.

The maintenance of this new class of systems is challenging at least for the three following reasons:

- *Heterogeneity.* The reduction of the *time to market* to cope with a competitive economic environment has forced software vendors to bet on the reuse of off-the-shelves components (*COTS*). That is, new software systems are build by assembling a set of existing software pieces developed by the vendor or his partners. Existing pieces of software are developed in various technologies. As a consequence, the maintenance of a software solution based on several technologies is very challenging for a human administrator.
- *Dynamicity.* Modern software systems evolve in a dynamic environment where resources they operate on are subject to constant changes. A change can be the availability or the unavailability of some services, or a workload surge in the demand of one service. Therefore, software systems need to be adapted constantly according to variations in their environment in order to achieve their objectives. When this dynamicity is coupled with scalability, it becomes very challenging for a human operator to manage them.
- *Scalability.* Software systems are distributed at large-scale, and consist of many interconnected components that collaborate in order to deliver the expected service. A global coordination of these components is challenging and error-prone.

Autonomic computing is a good candidate for addressing these challenges, because it enables software systems to manage themselves without or with a limited human intervention. Autonomic systems shift the burden of managing software systems from the human to

the system itself. The benefits of autonomic computing for the IT industry are multiple. For example, the reduction of the maintenance costs of software systems, the reduction of system failure due to human errors, or the optimization of resources usage. However, engineering such systems is a scientific and technological challenge.

Important progress has been made by the research and the industry community for developing autonomic solutions. Notably, the adoption of *service oriented architecture*(SOA) in order to cut down the complexity of engineering autonomic systems. SOA advocates a loose-coupling between components of a software system in order to manage the later efficiently. In addition, the increasingly automatization of administrative tasks in autonomic software solutions has reduced the workload for the human operator. Many autonomic systems provide high-level abstractions for hiding the complexity of the managed system. These abstractions enable developers of autonomic systems to express the behavior policies of autonomic systems without worrying about the complexity of the managed system.

Despite these progresses, engineering autonomic systems rise many difficulties. A prominent difficulty is the complexity of maintaining autonomic systems or making them evolve. That is because in traditional engineering approaches, the architecture of autonomic managers is masked under layers of abstractions. As a consequence, it is very difficult to understand the behavior of autonomic managers. This lack of visibility is a limitation for engineering large-scale autonomic systems where multiple autonomic managers collaborate together in order to adapt according to changes in their environment. In particular, the burden of detecting and preventing conflicts in the control architecture fall on developers' shoulders. This results in error-prone and time-consuming tasks for them.

This thesis takes a step ahead concerning the engineering of autonomic systems. We propose a new brand approach for engineering autonomic systems. Our approach proposes to reduce the time and efforts required for implementing autonomic systems. For that purpose, we rely on generative techniques of the code source and automated verifications tools for ensuring the validity of the control system architecture. Our approach fosters the visibility of the control system architecture and enables the implementation of amenable autonomic solutions.

Structure of the chapter

The rest of this chapter is organized as follows: In Section 1.2, we identify the motivation of this dissertation. Then, in Section 1.3, we introduce the goals of this dissertation. We give a glimpse of the contribution of this thesis in Section 1.4. Section 1.5 presents the structure of this document by reviewing main ideas of each chapter. Finally, we conclude this chapter with a list of publications (cf. Section 1.6).

1.2 Problem Statement

In the previous section, we have presented *autonomic computing* as a good candidate for tackling the growing complexity of software systems, in particular in a large-scale environment.

However, despite notable advances, many problems hinder the engineering of amenable autonomic systems. In this dissertation, we have identified some of them that we discuss in this section.

Visibility of the control system
Existing autonomic software solutions lack of visibility of the control system. In order to implement amenable autonomic systems, feedback control loops that govern self-adaptation must be visible at design and at runtime. This visibility increases the understanding of control flows, and enables a transparent coordination of feedback loops that implement various behavior. The visibility of the control system architecture is one of the problem towards amenable autonomic solutions. For many existing autonomic solutions, the control system architecture remains hidden at runtime.

Traceability from the control design to the runtime implementation
Traceability is a strong asset for having amenable autonomic solutions. In fact, very often the design and the implementation of the control system are not traceable. That is, it does not exist a strong coherence between the design and the implementation. That is because developers are not guided during the implementation of autonomic systems. As a consequence, developers implement control behavior in a adhoc manner. Traceability is important for supporting a flexible evolution of the control architecture.

Lack of tools and algorithms for verification
Large-scale autonomic systems, very often involve many control systems that collaborate to achieve conflictual objectives. The detection of this conflicts are crucial to ensure the consistency of the control system. In addition, the development of algorithms in order to check this conflict reduce the cost of engineering autonomic systems. The lack of visibility of the control architecture does not enable the implementation of tools and algorithms for verification in many existing autonomic solutions.

Transparent support of Non-Functional-Properties (NFP)
Autonomic solutions implement NFPs to customize the manner in which their functionalities are delivered. Many existing autonomic solutions does not provide support for implementing NFPs in a transparent manner, usually they are indistinguishably embedded in the control logic behavior. We think that in order to leverage a flexible adaptation of autonomic systems, NFPs must be transparently implemented in the control system.

1.3 Dissertation Goals

Our objective in this dissertation is to provide a solution for engineering amenable autonomic softwares. In the section above (cf. Section 1.2), we have highlighted some problems that hinder the achievement of that objective. In order to address these problems, we propose a generic approach for engineering autonomic solutions where feedback control loops are reified as first-class citizens. We leverage the visibility of feedback loops by providing a control-oriented language for designing control systems. We address the traceability from

the design to the implementation of the control architecture by ensuring a strong mapping between design and runtime concepts. The visibility of the control system enables to develop algorithms for automatic checking of the control architecture. Finally, we provide a flexible support for the implementation of non-functional-properties in the control architecture. The main goals of our proposal are the following:

- **Domain agnosticism.** We aim at defining a generic solution which targets several application domains. Agnosticism with regards to the domain of application enables our solution to be used for a wide range of autonomic systems. The design of the control architecture must be independent of a specific implementation platform.
- **Flexible adaptation and evolution.** To reduce the complexity of maintaining autonomic systems, feedback loops that govern adaptations must become visible at runtime. The visibility of control flows in the control system facilitates the engineering of scalable adaptive systems where several control systems collaborate in order to achieve complex objectives. In addition, it facilitates the evolution of the control system architecture.
- **Cost-effectiveness.** It is important to develop a solution that is cost-effective for engineering autonomic systems. The cost of engineering autonomic system can be addressed at two different levels: At the implementation level, by providing an implementation support for developers of autonomic systems; at the design level by supporting automatic verification of the control architecture.
- **Consistency.** The consistency of the control system architecture is a good property for building reliable autonomic systems. It is important to detect potential conflicts in the control system architecture before the deployment. This will save a lot of time and efforts for developers of autonomic systems.

1.4 Contributions

In order to enhance the understanding of this dissertation, we briefly describe in this section the main contribution of this thesis. The contribution of this dissertation can be organized in the four following points:

- The first contribution is the reification of feedback control loops as first-class citizen at runtime. This is done by ensuring a strong mapping between architectural concepts and implementation concepts. We leverage the visibility of feedback control loops by relying on a service component platform.
- The second contribution consists of enabling the customization of the MAPE-K architecture pattern of feedback control loops. In particular, we focus on the integration of stabilization algorithms for designing stable control systems.

- The third contribution is the traceability. we provide an automated mapping between the control architecture design and its runtime implementation. The implementation of the control architecture is guided in order to ensure the conformity between the design and the implementation.

- The fourth contribution is the support for verification when building autonomic systems. We have identified and characterized conflicts patterns that can appears during control systems design, and have implemented tools for their detection and the automatic resolution of some them.

1.5 Dissertation Roadmap

This dissertation is divided in four parts. The first part gives an overview of the state-of-the-art of autonomic solutions, and introduces the SALTY[1] meta-model which is used in this dissertation for explaining our contribution. The second part is the *Contribution part*. This part explains the proposal of this dissertation. The third part of this thesis is the *Validation part*. The *Validation part* presents some experimental evaluations that validate the proposal of this dissertation. The *Conclusion part* is the fourth part of this dissertation. It provides a summary of this work, and discusses some perspectives.

Part I: State of the Art

- **Chapter 2: State of the Art.** In chapter 2, we introduce some backgrounds and definitions related to autonomic systems. Then, we give an overview of architecture-based autonomic solutions, classify them, and point out some limitations of these solutions.

- **Chapter 3: SALTY Model.** This chapter presents the SALTY meta-model. In chapter 3, we first introduce the graphical formalism for designing a control architecture with the SALTY meta-model.

Part II: Contribution

- **Chapter 4: Contributions Overview.** This chapter revisits the challenges and the objectives of this dissertation. Then, an overview of the our contribution called *CORONA* is introduced.

- **Chapter 5: Runtime Architecture.** This chapter presents two contributions of this dissertation. The first contribution consists of reifying feedback control loops as first-class citizen at runtime. The second contribution depicted in this chapter is the support for non-functional-properties through the customization of the control loop architecture.

[1]This thesis was funded by the SALTY ANR project in which I participated. https://salty.unice.fr/

- **Chapter 6: Compilation Infrastructure.** This chapter discusses the *CORONA* toolchain that provides support for engineering cost-effective autonomic system. It covers two other contributions of this dissertation. The first one is the traceability between the design and the implementation of control system, and the second one is the verification algorithms for conflict checking in the control architecture.

Part III: Validation

- **Chapter 7: Condor Case-Study.** This chapter validates one of the claim of this thesis which is the *consistency* of the control architecture. It shows how verifications algorithms can be used for designing more reliable autonomic systems. This is illustrated on the Condor case-study.
- **Chapter 8: Fire Emergency Case-Study.** This chapter validates the second claim of this dissertation which is *integration & transparency*. In particular, we discuss how control systems developed with *CORONA* can be integrated in existing control architecture. The *integration & transparency* claim is illustrated with the *Fire emergency* case-study.
- **Chapter 9: Smart Mall Case-Study.** This chapter validates the integration of stabilization algorithms in order to design flexible control systems

Part IV: Conclusion

- **Chapter 10: Conclusion.** This chapter gives a summary of this dissertation and recall the contributions of our proposal. We conclude this dissertation with a discussion of the perspectives of our work. We point out interesting research directions that could be addressed to complete and improve this thesis.

1.6 Publications

Below, we present the list of research articles that were published during this thesis.

International Journals

- *The DigiHome Service-Oriented Platform*. Daniel Romero, Gabriel Hermosillo, Amirhosein Taherkordi, Russel Nzekwa, Romain Rouvoy and Frank Eliassen. Software: Practice and Experience (SPE). 2011. *Rank*(CORE): A. *To appear*.

International Conferences

- *RESTful Integration of Heterogeneous Devices in Pervasive Environments*. Daniel Romero, Gabriel Hermosillo, Amirhosein Taherkordi, Russel Nzekwa, Romain Rouvoy and Frank Eliassen. In 10th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS'10). pages 113–126 of LNCS 6115 (Springer). Amsterdam, Netherlands. June 2010.

International Workshops

- *Modelling Feedback Control Loops for Self-Adaptive Systems*. Russel Nzekwa, Romain Rouvoy, Lionel Seinturier. 3rd International DisCoTec Workshop on Context-Aware Adaptation Mechanisms for Pervasive and Ubiquitous Services (CAMPUS 2010). Amsterdam, Netherlands. June 2010.
- *A Flexible Context Stabilization Approach for Self-Adaptive Application*. Russel Nzekwa, Romain Rouvoy, Lionel Seinturier. 7th IEEE Workshop on Context Modeling and Reasoning (CoMoRea) at the 8th IEEE International Conference on Pervasive Computing and Communication (PerCom'10). Manheim, Germany. March 2010.
- *Towards a Stable Decision-Making Middleware for Very-Large-Scale Self-Adaptive Systems*. Russel Nzekwa, Romain Rouvoy, Lionel Seinturier. The 8th BElgian-NEtherlands software eVOLution seminar (BENEVOL'09). Louvain-la-Neuve, Belgium. December 2009.

Part I

State of the Art

Chapter 2

State of the Art

"Nothing in the world is more dangerous than sincere ignorance and conscientious stupidity."
–Martin Luther King

Contents

2.1 Autonomic Computing 12
2.1.1 Context 12
2.1.2 Definitions & Taxonomy 12
2.1.3 Autonomic System Properties 15
2.2 Existing Autonomic Systems Approaches 16
2.2.1 JADE 16
2.2.2 TUNe 18
2.2.3 UNITY 18
2.2.4 AUTONOMIA 20
2.2.5 RAINBOW 21
2.2.6 CEYLON 22
2.2.7 DiaSpec 24
2.3 Assessing Autonomic Systems 26
2.3.1 Comparison Criteria 26
2.3.2 Classification of Autonomic System Approaches 27
2.4 Summary 30

In this chapter, we introduce some basics concepts and software approaches used as a background for this dissertation, we also present the scientific context and rationales of this thesis. In particular, we define concepts related to autonomic computing and review existent software solutions for building autonomic systems.

Structure of the Chapter

This chapter is organized as follows: Section 2.1 defines and introduces some concepts related to autonomic computing. Section 2.2 presents existing software approaches for building autonomic systems. In Section 2.3, we discuss the limitations of these approaches and point out challenges ahead in order to address them. Finally, we conclude (cf. Section 2.4) this chapter with a summary.

2.1 Autonomic Computing

In this section, we start by introducing the context of autonomic computing (cf. Section 2.1.1), then we give some definitions and explain some concepts related to the autonomic computing (cf. Section 2.1.2). We end this section by presenting some properties of an autonomic control system (cf. Section 2.1.3).

2.1.1 Context

In the previous chapter (cf. Chapter 1), we have stated that the increasing complexity of software systems, and their maintenance costs were the prominent motivations for autonomic computing. *Autonomic computing* draws upon a wide range of research fields, from biology to robotic through control engineering [Kep05]. This makes difficult any attempt for characterizing and describing this research field.

The term *autonomic* in *autonomic computing*, comes from the analogy with the autonomous nervous systems that ensures the regulation of visceral functions of the human body like, the respiration, the digestion or the micturition. The nature provides plenty of illustrations of complex autonomic systems that cooperate in an astonishing manner in order to achieve the desired goal. Despite the complexity of biological autonomic systems, they appear to be more tolerant to failure than software systems. One of the purpose of *autonomic computing* is to identify and understand principles behind the self-adaptive behavior of natural autonomic systems, in order to implement software systems with similar properties.

In contrast to biological *autonomous systems* where control loops that govern self-adaptation are often not visible, feedback loops are the central element of control engineering. Control engineers are provided with well established models, tools and techniques for designing and characterizing control systems [DB00]. *Autonomic computing* researchers seek to which extent some principles of control theory can be used for reasoning about autonomic software systems.

2.1.2 Definitions & Taxonomy

There are more than one definition of what is *autonomic computing* in the literature. We have retained here three of them, proposed by renowned scientists of this research field:

Definition 1: "The vision of autonomic computing is to create software through self-* properties" [SH05].

Definition 2: "Autonomic computing is the ability to manage your computing enterprise through hardware and software that automatically and dynamically responds to the requirements of your business" [Mur04].

Definition 3: "Systems manage themselves according to an administrator's goals. New components integrate as effortlessly as a new cell establishes itself in the human body" [Kep05].

The above three definitions are coherent, and underline the ultimate purpose of autonomic computing, which is to reduce the complexity of implementing and maintaining information systems for software engineers.

Autonomic computing vision for addressing the complexity of software systems premises on the *IBM* blueprint [IBM01]. According to the *IBM's* vision, *autonomic systems* must be a collection of interacting *autonomic elements*. *Autonomic elements* are the individual constituents of the system that deliver a given service to humans or other *autonomic elements*. *Autonomic elements* must manage their relationships with their environment (managed system or other autonomic elements) in accordance with established policies of the autonomic system. Consequently, they will relieve humans from the responsibility of directly intervene for adjusting the behavior of the managed system. The *IBM's* vision of autonomic systems is modeled into the *MAPE-K* paradigm.

Figure 2.1 depicts the structure an *autonomic element*. The *autonomic element* implements the *MAPE-K* paradigm. *MAPE-K* stands for *Monitor*, *Analyze*, *Plan*, *Execute* and *Knowledge*. The *MAPE-K* paradigm defines the essentials activities of an autonomic element.

Monitor. The feedback cycle of an autonomic element starts with the monitoring activity. This activity consists of collecting information from the environment of the autonomic element that reflect changes of the monitored system. The monitoring activity is implemented by *sensors*.

Analyze. An autonomic element analyzes collected data from the previous activity (monitoring). The activity *analyze* is a part of the reasoning or decision-making of the autonomic element and aims at identifying symptoms that require the system to take specific actions.

Plan. The *plan* activity collects information from the activity *analyze* and makes decision about how the monitored system need to be adapt in order to reach the desirable state.

Execute. To implement the decision taken, the autonomic element interacts with the monitored systems through *effectors* or *actuators*.

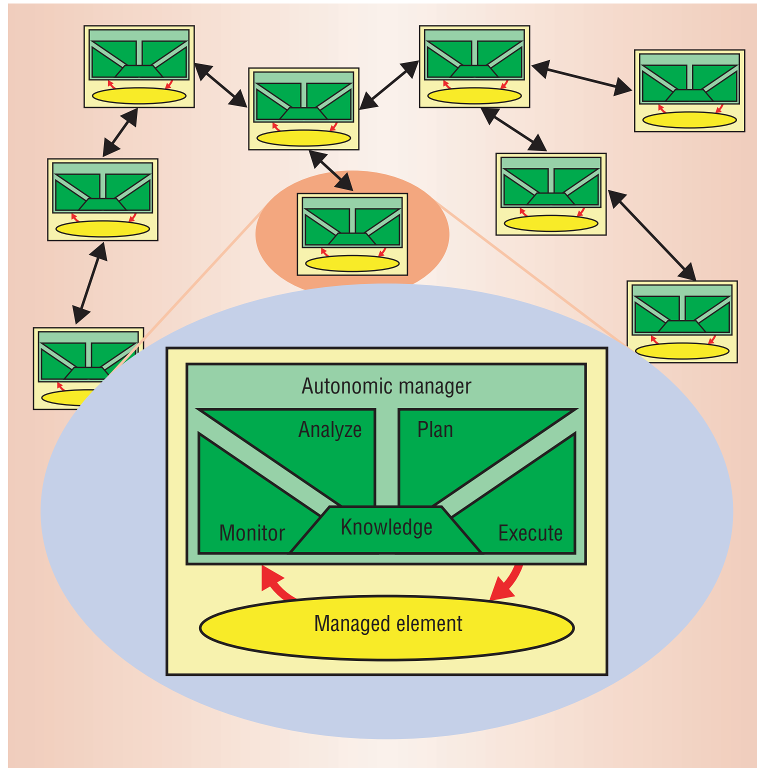

Figure 2.1: Structure of an Autonomic Element [Kep05]

Knowledge in the *MAPE-K* paradigm does not correspond to an activity of the autonomic element. It expresses the base of information (control flow) that is exchanged between the four activities of an autonomic element. *Autonomic elements* are also known as *autonomic managers* because they regulate the behavior of the system that they control by interacting with the later. In the IBM vision of autonomic computing, an autonomic system arises from the interactions of autonomic managers amongst themselves (the coordination), and their interactions with the managed system. Autonomic elements of an autonomic system form the *control system*. An autonomic manager can be designed for achieving one or several goals. The term *feedback control loop* (FCL) usually refers to a control flow of the autonomic manager that implement a single goal. The *control flow* denotes the sequence of actions corresponding to activities of an autonomic manager.

Figure 2.2 depicts an autonomic system architecture. It generally consists of two subsystems: the control system that implements the control behavior, and the managed system that consists of managed elements interacting with the control system. The control system consists of autonomic managers that cooperate in order to ensure a consistent behavior of the autonomic system. The type of coordination between autonomic managers can be of various forms [ASHP+08], *hierarchical*, *peer-to-peer*, *hybrid*. Interactions between autonomic managers and managed elements take place through touch-points.

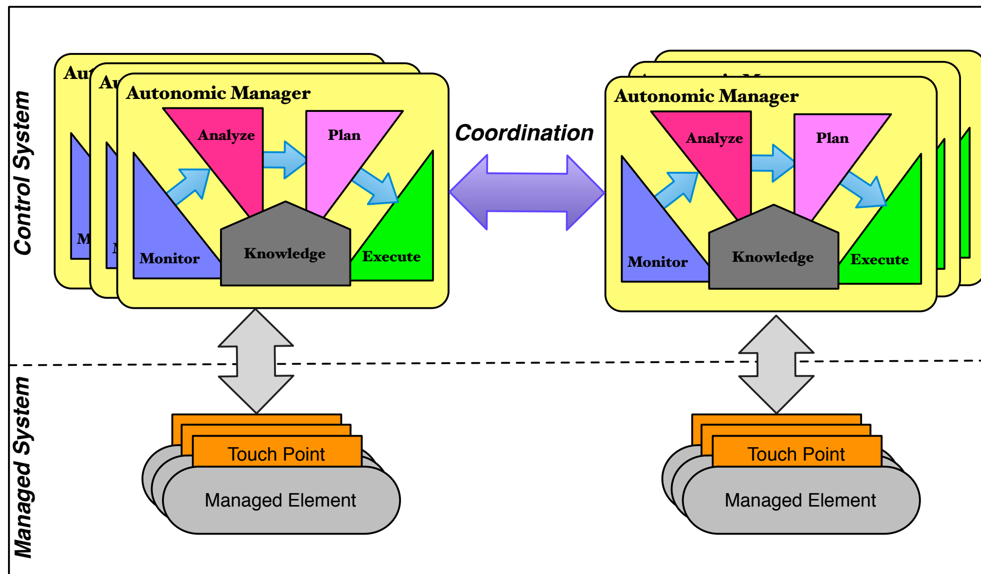

Figure 2.2: Overview of an Autonomic System Architecture

In this section, we have introduced some definitions and concepts related to autonomic computing. We have also depicted the structure of an autonomic element with respect to the MAPE-K paradigm promoted by IBM. Autonomic systems adapt their behaviors according to changes in their environment in order to conform to their goals. Consequently, adaptation is a central aspect for autonomic systems. These adaptations can be of many kinds. In the next section, we introduce some properties of autonomic systems from the adaptation point of view.

2.1.3 Autonomic System Properties

Autonomic systems are characterized by four properties that defined their behavior. These properties were defined in accordance to self-adaptation mechanisms in biological organisms [KC03]. An autonomic system that implements all these four properties is qualified as *self-managed* system. The following list elaborates in details on the properties of autonomic systems.

- *Self-configuration.* This property refers to the capability of an autonomic system for dynamic reconfiguration in response to changes in the environment by installing, updating, or integrating new software entities.
- *Self-healing.* This property is also know as *self-repair*. It refers to the capability of an autonomic system to discover, diagnose, and react to disruptions. This property encapsulates proactive actions of the system for preventing errors, faults or failures.

- *Self-optimization.* This property is also known as *self-tuning*. It characterizes the capability of the system for managing performance and resource allocation in order to satisfy its goals.

- *Self-protecting.* This property refers to the capability of the system for detecting failures and recover from them. *Self-protecting* autonomic systems defend their-selves against malicious attacks and mitigate their effects on the system behavior.

The above properties are usually subsumed under the term of *CHOP* (Configure, Heal, Optimize, and Protect) [HS06] or self-*. Now that we have introduced the architecture and properties of autonomic systems, let us look in details into some existing autonomic solutions.

2.2 Existing Autonomic Systems Approaches

As we mentioned at the beginning of this chapter, *autonomic computing* embraces a wide range of research fields, and makes it difficult to provide an exhaustive vision of the state-of-the-art. In this section, we present existing approaches for building autonomic systems. However, we are going to focus the spectrum of our analysis to *architecture-based approaches* which present a great interest for the rest of this dissertation.

Architecture-based approaches define for a class of systems, a vocabulary of element types, properties common to the element types in theses systems, a set of constraints on the permitted composition, and the associated analyses for reasoning about this class of systems [AAG93]. This means that *architecture-based approaches* provide a higher level abstraction for reasoning or implementing solutions for a given domain.

2.2.1 JADE

Jade [BPHT06] is a framework for the management of distributed autonomic systems. Jade focus on the management of *J2EE* cluster legacy applications. Jade relies on the FRACTAL [BCS02] component model. The FRACTAL component model is a generic model for implementing complex software systems that can be dynamically adapted.

The autonomic behavior is achieved through autonomic managers that are in charge of regulating the behavior of the managed system. Figure 2.3 gives an overview of the Jade approach. The implementation of the control system and the interaction with the managed system is done as follows:

- **Interaction with the Managed System**
  Since managed applications are heterogeneous, developers must implement wrappers as FRACTAL components in order to provide uniform interfaces to interact with the control system. That is, the legacy system layer (clustered applications) is wrapped into a management layer implemented by FRACTAL components.

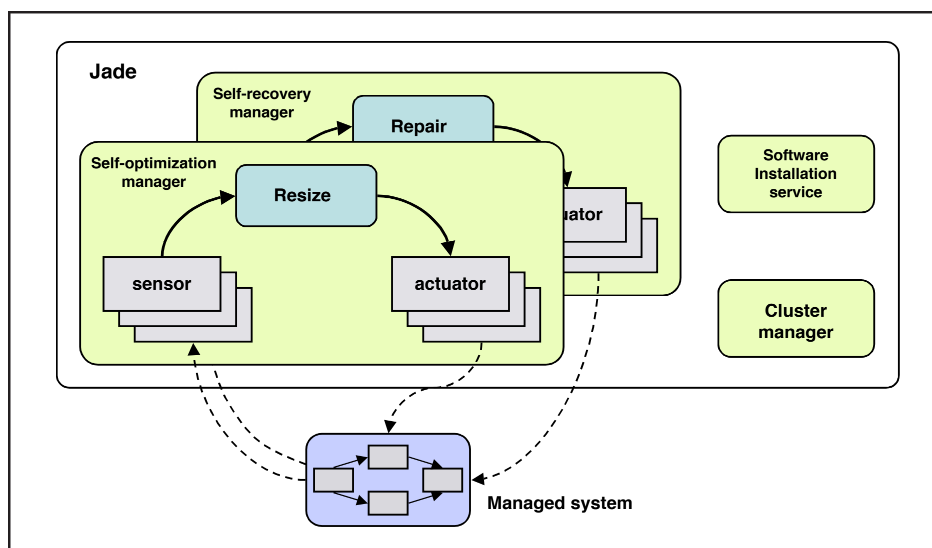

Figure 2.3: Overview of the Jade Approach [BPHT06]

- **Implementation of the Autonomic Control Loop**
  Autonomic managers in the Jade approach are organized according to self-properties aspects like *self-optimization* or *self-healing*. Autonomic managers are implemented according to the MAPE-K paradigm. Jade provides flexible mechanisms for creating autonomic managers and to manage their deployment. The logic behavior of an autonomic manager is implemented in the decision block of the autonomic manager. It can be for example, an algorithm for resizing the cluster of replicated servers upon load surges for a self-optimization autonomic manager.

In the Jade approach, the implementation of autonomic managers conforms to the MAPE-K model. However, the separation of autonomic managers according to self-properties aspects is not always possible, because sometimes self-adaptive behaviors overlap, and cannot be represented independently. In this case, it is almost impossible to implement them independently without creating conflicts. In addition, in the Jade approach, the coordination between autonomic managers is not explicit. Finally, Jade uses a generic syntax language (FRACTAL component syntax) for describing and implementing the architecture of the control system. As a result, the visibility of the control system is hidden at runtime because it is hard to distinguish between control architecture components and managed system components. This lack of visibility is a threat for the autonomic system, because it does not allow verifications of the control system architecture in order to detect potentials conflicts between autonomic managers.

2.2.2 TUNe

Tune [BHS+08] is a platform designed for building autonomic managers. Tune is an extension of the Jade framework. As Jade, Tune focuses on the autonomic management of legacy systems. Tune introduces an higher-level formalism for the specification of the deployment and the management policies of legacy J2EE systems. The purpose of the higher-level formalism is to hide details of the FRACTAL component model, and provide a more intuitive syntax for wrapping these systems into a uniform abstraction interface, than what was available in Jade.

The implementation of the control system and the interaction with the managed system is done as following:

- **Interaction with the Managed System**
  In Tune, interactions with the managed system are implemented through the WDL language. This language enables developers to specified a list of *methods* with parameters, that can be called for a given *wrapper* component. At this stage the notion of *sensor* and *effector* is not explicit.

- **Implementation of the Autonomic Control Loop**
  The implementation of the control behavior is basically done through the reconfiguration language where developers specify reconfiguration policies for an autonomic manager. However, like for Jade, it is not clear how the control logic of an autonomic manager can be changed or specified.

Overall, Tune provides a higher level formalism to facilitate the implementation of autonomic managers for legacy J2EE servers. However, the introduction of a supplementary layer of abstraction masks the visibility of the control system architecture. This lack of visibility is a threat for implementing consistent control systems.

2.2.3 UNITY

Unity [CSW04] is one the first autonomic software solution that was implemented. Unity was developed by IBM which is a pioneer company in the development of software solutions that enable to build effective autonomic systems. Concurrently to the Unity project, IBM has developed other projects like *ABLE* [BSP+02] or the *Autonomic Toolkit* [Bar04] that rely on the multi-agent paradigm for implementing autonomic behavior.

Unity was designed for managing resource allocation in a distributed system [TCW+04]. The Unity project relies on the *MAPE-K* paradigm. Figure 2.4 illustrates the architecture of the Unity autonomic system. The Figure shows that the architecture of Unity autonomic system consist of *autonomic applications*, a *resource pool*, an *arbiter*, and a *policy repository*.

The implementation of the control system and the interaction with the managed system is realized as following:

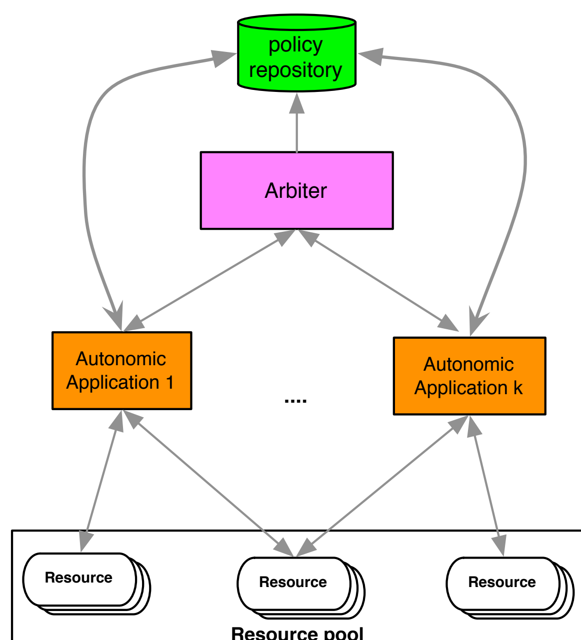

Figure 2.4: Architecture of Unity Autonomic System

- **Interaction with the Managed System**
  The managed system in Unity is represented by the *resource pool*. *Autonomic applications* interact through managers directly with the *resource pool*. Mangers are able to increase or decrease resource allocation for a given application according to its needs. The communication between *autonomic applications* and the *resource pool* is completely specified and hard-coded through Java interfaces. The notions of *sensor* and *effector* remain very implicit.

- **Implementation of the Autonomic Control Loop**
  The implementation of the control system is realized at the level of each application through a *manager*. The logic behavior of a *manager* is specified through a high-level policy language. All policies defined in the Unity environment are referenced in the *policy repository*. The resource *Arbiter* is used in order to coordinate the access of *autonomic applications* to the *resource pool*.

Unity provides a very specialized and flexible solution for building autonomic systems. It focuses on the issue of resource allocation. Unity allows developers to specify control policies with an high level syntax. However, the specialization of Unity makes difficult its reuse for other type of autonomic management not related to the resource allocation. In addition, the use of an *arbiter* for the coordination of *autonomic applications* limits the scalability of Unity. Finally, although the MAPE-K paradigm is used in Unity, the architecture of the control system remains fuzzy because its constituents are not clearly identified. For example,

the notions of *sensor* and *effector* of the MAPE-K paradigm cannot be explicitly infered from interfaces implementing the communication between the *managed* and the *control system*.

2.2.4 AUTONOMIA

The *Autonomia* [XSL+03][YCHP05] project was developed at the University of Arizona. *Autonomia* defines an approach for building autonomic systems on the basis of multi-agents. The purpose of the project is to provide developers of autonomic systems with tools for implementing mobile autonomous agents. *Autonomia* focus on increasing capacities of existing software applications with "agent properties". Figure 2.5 depicts the architecture of an autonomic agent in *autonomia*.

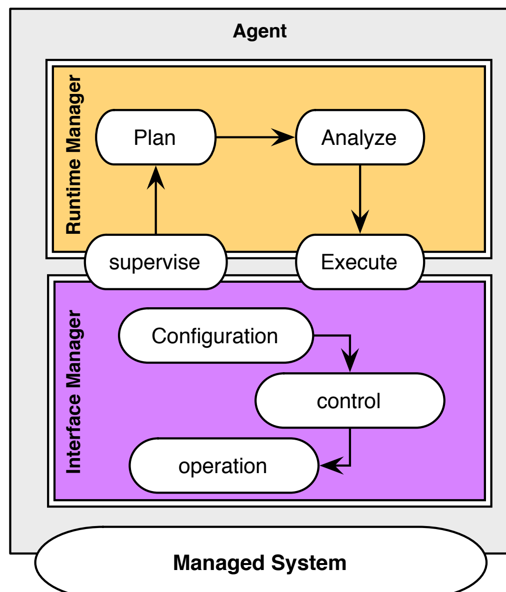

Figure 2.5: Architecture of an Autonomic Agent in Autonomia

Figure 2.5 shows that the architecture of an autonomic agent consists of two layers: the interface manager and the runtime manager. These layers implement the autonomic behavior of an agent in *autonomia*. In *autonomia*, the implementation of the control system and the interaction with the managed system is realized as follows:

- **Interaction with the Managed System**
  Interactions with the managed systems are organized through the *interface manager*. It defines a set of operations that can be performed on the managed systems. These operations can be supervised by controllers which can be configured for executing autonomic tasks. The main constituents of the *interface manager* are depicted on Figure 2.5.

- **Implementation of the Autonomic Control Loop**
  The implementation of the control behavior within an agent is realized by the *runtime manager*. The *runtime manager* defines the management policy of the autonomic agent. These policies are expressed in the form of *event-condition-action* rules. As depicted on Figure 2.5, the architecture of the runtime manager conforms the MAPE-K paradigm.

The whole approach adopted in Autonomia for implementing autonomic systems is very close to the one promoted by ORACLE with the JMX [ORA] framework in Java. JMX offers the possibility to enhance Java applications with management capabilities through the extension of JMX's interfaces by Java classes. The architecture of the Autonomia framework is highly distributed and bring a lot of flexibility for defining policies of an autonomic agent. However, the definition of these policies are not coordinated for all autonomic agents of the environment. This can lead to some inconsistencies of the control system architecture. Finally, the language for implementing the control system does not reify its architecture.

2.2.5 RAINBOW

Rainbow [wC08][wCcHG+04] is a framework for implementing autonomic systems developed at Carnegie Mellon University in 2004. The main claim of Rainbow is to reduce the cost of implementing autonomic systems. For that purpose, Rainbow defines a generic architecture of an autonomic system based on the MAPE-K paradigm that developers can customize to their needs. Rainbow supports a high-level description language for implementing the behavior of the control system. Figure 2.6 depicts the architecture of the Rainbow autonomic system.

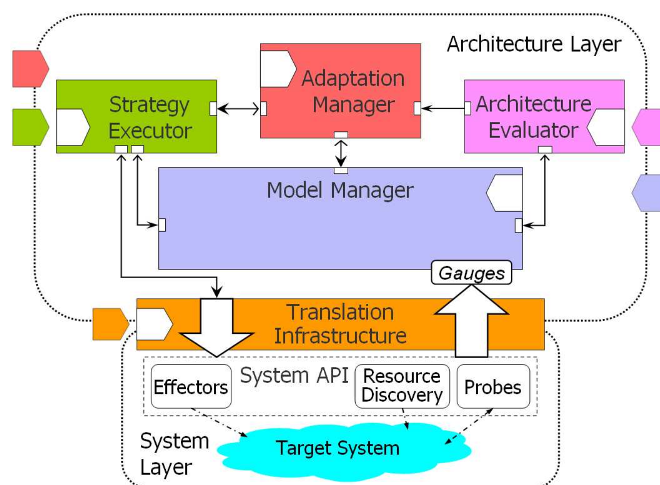

Figure 2.6: Rainbow System Architecture with Customization Points [wC08]

Figure 2.6 shows that the Rainbow architecture conforms to the MAPE-K paradigm. The customization of the Rainbow architecture is represented by colored pentagons on Figure 2.6. In Rainbow, the implementation of the control system and the interaction with the managed system is realized as follows:

- **Interaction with the Managed System**
  Rainbow requires a model of the managed system for reasoning about changes in the system. The translation infrastructure plays the role of mapping between the model of the managed system and its implementation. The translation infrastructure interacts with the managed system through effectors and probes. It forwards the informations collected from the managed system to the autonomic control system.

- **Implementation of the Autonomic Control Loop**
  The implementation of the control system in Rainbow requires to define the model of the managed system in addition to the control architecture model. The constituents of the control architecture exist in a generic form, but must be customized for implementing specific behavior of a given application. The description of the autonomic manager in Rainbow is independent of the execution platform.

Rainbow offers an effective way to reduce implementation efforts when building autonomic systems. However, the main limitation of the Rainbow approach as pointed by the authors [wC08] is the centralization of the control system which can be an issue for the scalability of the solution.

2.2.6 CEYLON

Ceylon [MDL10] is a service oriented framework for building autonomic managers. Ceylon implements autonomic managers like an opportunistic composition of service oriented components. In Ceylon, complex management policies are achieved through the orchestration of simple policies implemented by specialized components. Figure 2.7 depicts the architecture of Ceylon autonomic system.

Figure 2.7 shows that the behavior of the control system is the result of the composition of service components by the *task manager*. The composition is opportunistic in the sense that, it is performed at runtime depending on the high level goals of the system. The implementation of the control system and the interaction with the managed system is realized as follows:

- **Interaction with the Managed System**
  Interactions with the managed system are implemented by individual service components. Components are activated or stopped by the task manager which is in charge of coordinating the behavior of individual components in order to achieved the goal

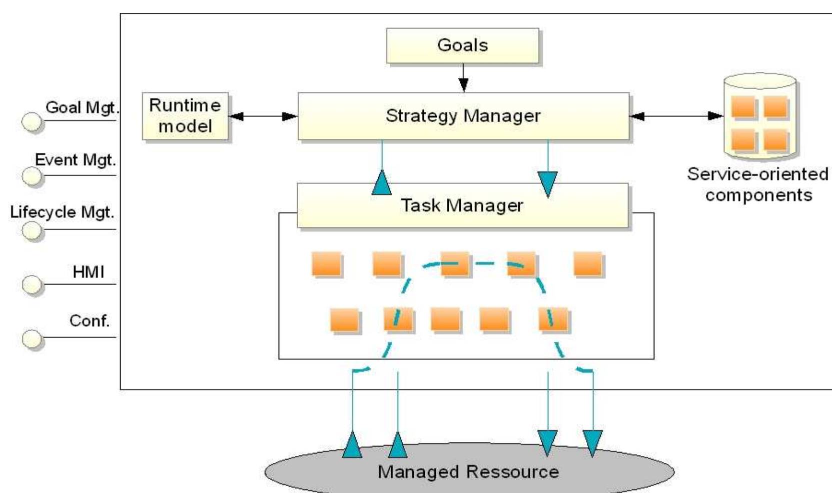

Figure 2.7: Ceylan Autonomic Control Architecture [MDL10]

of the control system. Ceylon relies on OSGI/iPOJO components model for the implementation behavior of service components. The communication layer between service components and the managed system is not explicit for developers, neither the implementation of these components. In the Ceylon approach, service components are provided by the platform and registered in a repository as shown on Figure 2.7.

- **Implementation of the Autonomic Control Loop**
  As depicted in Figure 2.7, the control system in Ceylon is implemented by two entities: the strategy manager and the task manager. The strategy manager supervises the task manager. The task manager orchestrates the behavior of service components, by activating and deactivating them when necessary. The behavior of the control system is configured through interfaces of the Ceylon container.The Ceylon container provides high level syntax for customizing different aspects of the control behavior like goal management, life-cycle management or human-machine interactions.

The strength of Ceylon is the service oriented component (SoC) approach for implementing autonomic systems. SoC offers an effective way to address the scalability of autonomic systems. In addition, Ceylon provides a high level syntax for specifying the behavior of the control system. However, the visibility of the control system architecture is masked by multiple layers of abstractions. In particular, interactions between service components and the managed system, as well as their implementation logic are not transparent for developers. This is a critical threat for the control system because, it is a potential source of conflicts. The task manager tries to limit conflicts by ensuring that components with similar behavior are

not activated at the same time. However, since the architecture of the later is not explicit, the consistency of their coordination cannot be guaranteed.

2.2.7 DiaSpec

DiaSpec [CBCL11] is a domain specific architectural framework dedicated for the implementation of autonomic applications. DiaSpec promotes the SCC (*sense-compute-control*) pattern. The SCC pattern presents a great similarity with the MAPE-K pattern. From an architectural perspective, in the SCC paradigm, we distinguish four types of elements: *(1) sensors* at the bottom of the architecture collect information from the environment; then *(2) context operators* which processed this information; then *(3) control operators*, which refine this information; and finally *actuators* at the top of the architecture which impact the environment. The Architectural style of the DiaSpec framework is depicted in Figure 2.8.

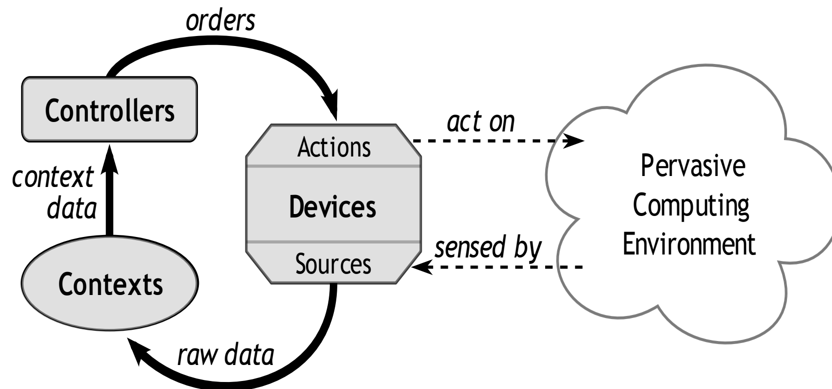

Figure 2.8: Control Architectural Style in DiaSpec

Figure 2.8 shows the SCC organization of the control architecture in DiaSpeC. One of the specificity of DiaSPec is that, it uses generative techniques (MDE) [BG01] for generating the implementation framework of the control system from an initial architectural description. The implementation of the control system and the interaction with the managed system is realized as follows:

- **Interaction with the Managed System**
  From the architectural description of the control system, diaspec generates an implementation framework. The implementation framework enables to specify the logic behavior of each elements of the control system. Interactions with the managed system are implemented from sensors and effectors classes provided by the implementation framework. On Figure 2.8, sensors correspond to *sources* of devices and effectors to *actions* of devices.

- **Implementation of the Autonomic Control Loop**
  The behavior of the control system is implemented from the generated implementation framework. This is done by extending abstract classes corresponding to the control architecture elements (sensors, operators, effectors).

The DiaSpec approach enables to build autonomic system in three phases: The first phase consists in the description of the control system architecture through a domain specific architectural language. The second phase consists in the implementation of the behavior of the control system through a generated implementation framework language. The third phase is the distribution phase which consists in exporting the implemented control system into different target technologies like *RMI*, *SIP* or *web services*. The generative approach adopted by DiaSpec significantly reduces the burden of implementing control systems for developers, while providing a high flexibility for adapting it to their needs. The visibility of the control architecture is raised through the explicit support of the SCC paradigm in the implementation framework.

One limitation of DiaSpec lies in the management of cross-cutting concerns. In particular, the management of conflicts between feedback loops involved in the control architecture is not addressed. A recent work [JCL11] based on DiaSpec addresses this issue by providing and extension language for implementing architecture conflicts of the control system. In addition, in DiaSpec, the chain of visibility of the control system is interrupted at runtime because the transformation from the implementation framework to the target runtime platform does not guarantee a strong mapping between architectural concepts and their implementation at runtime.

Summary

In this section, we have overviewed the landscape of architecture-based autonomic solutions. We have noticed that existing solutions are very diverse, with their own strengths and weaknesses. In particular, we have showed that some solutions like *Rainbow*, or *Tune* require a model of the managed system in order to implement autonomic applications. We have noticed that many paradigms are used to structure the architecture of the control system, and namely, the *SCC* paradigm, the *MAPE-K* paradigm or *agent-based* paradigms. We have presented solutions that rely on centralized control architecture, and others that are fully decentralized like *Ceylon*. We have presented the advantages of an higher customization of the control architecture in terms of reuse of architectural components, and the flexibility provided by a generative approach. At this stage, it appears that in order to clearly identify challenges ahead for improving existing solutions of the state-of-the-art, it is relevant to compare these solutions on the basis of common criteria. That is what we are going to focus on in the next section.

2.3 Assessing Autonomic Systems

In this section, we introduce some criteria in order to classify autonomic system solutions (cf. Section 2.3.1) presented in the previous section. Then, we present a classification (cf. Section 2.3.2) of these solutions in order to leverage challenges to address for improving the engineering of software autonomic systems.

2.3.1 Comparison Criteria

Autonomic systems can be classified according to two dimensions: *Quality/Quantity dimension* and *engineering dimension*.

Quality/Quantity Dimension

The *quality/quantity dimension* refers to criteria used to assess autonomic solutions in a quantitative or qualitative manner. In [MH04], McCann et al. suggest the following metrics for assessing autonomic solutions according to *quality/quantity dimension*: *Quality of Service, cost, stabilization, sensitivity, adaptation and reaction time*.

- *Quality of Service* is a metric which evaluates to which extend the system has reached its primary goals. It can consist of elementary metrics like the response time over the service execution cost. It is an highly important metric for autonomic systems as they are typically designed for improving some aspects of a service.
- *Cost* is a broad metric which evaluates the cost of the autonomicity. The cost can be measured in terms of the amount of communication between components of the control system, or in terms of time and people required for developing and maintaining an autonomic solution.
- *Stabilization* is a metric which refers to the sensitivity of the system, and evaluates the time taken for an autonomic solution to learn its environment and stabilize its operations.
- *Sensitivity* is a metric which measures how well an autonomic system fits with the environment in which it evolves. At one extreme, an highly sensitive system will detect subtle changes in the environment and adapt according to. At another extreme, lowly sensitive system will not be able to detect important changes of the environment. Depending on the type of activities, a trade-off between both extremes must be found to have an efficient behavior of the system.
- *Adaptation and Reaction Time* metrics are related to the system reconfiguration and adaptation. The adaptation time measures the time that it takes for the system to adapt to a change in the environment. The reaction time measures the time between when an environmental element has changed and the system recognizes that change. The reaction time can be seen as a part of the adaptation time.

The comparison between autonomic solutions through *quality/quantity dimension* criteria required the use of benchmark applications, to ensure that autonomic systems are evaluated within the same context. So far, a consensus does not exist concerning benchmark applications for the evaluation of autonomic systems. For that reason, we will focus our comparison on engineering dimension criteria.

Engineering Dimension
The *engineering dimension* refers to the criteria used to characterize autonomic system in a descriptive manner. *Engineering dimension* defines several aspects that characterize the variability in the engineering process of autonomic systems. *Engineering dimension* is related to design and runtime choices in the engineering process of autonomic systems. To classify existing autonomic solutions according to the engineering dimension, we will use the following criteria:

- *Abstraction Layer.* This criteria refers to the level of abstraction provided by an autonomic solution for implementing the control system. The abstraction layer can be *High*– general purpose or domain specific–, or *Low*–general purpose or domain specific. An *high* abstraction layer means that an autonomic solution provides an high level architectural language for implementing the control system.
- *Customization/Reuse.* This criteria characterizes the support for reusing customizable parts of the control system architecture. That is, when implementing a new control system, developers can benefit from existing pieces of the control system.
- *Scalability.* This criteria refers to the centralized or decentralized coordination of feedback control loops implemented with an autonomic solution. It characterizes the suitability of the control system for managing large scale distributed infrastructures.
- *Autonomic Properties.* This criteria expresses the possibility of implementing all self-adaptive behaviors of an autonomic system. It tells whether an autonomic solution is specialized or not for the implementation of a specific self-adaptive behavior.
- *Control Visibility.* This criteria defines the visibility of the constituent elements of the control system. The control visibility is characterized at design and at run time.

2.3.2 Classification of Autonomic System Approaches

In this section, we introduce the classification of common architecture-based autonomic solutions presented above (Section 2.2). This classification is based on the engineering dimension criteria. To ease the readability of this classification, we adopt the following semantics. The *yes* denotes that the criteria is fully implemented by the autonomic solution; the *yes/no* means that the criteria is partially implemented by the autonomic solution; and the *no* denotes that the criteria is not implemented for the autonomic solution. *High or low* denotes that an autonomic solution provides a high or low level abstraction language. *Generic or Domain-specific*

indicates that the language used for implementing autonomic behaviors is general-purpose or specific to an application domain. *Self-CHOP* indicates the kind of self-adaptive property that can be implemented with an autonomic solution. Each Letter of *CHOP* indicates one self-adaptive property, and namely Configuration, *H*ealing, *O*ptimization, and *P*rotection. Table 2.1 gives the classification of architecture based solutions with respect to the engineering criteria.

Architecture-based Autonomic solutions	**Classification Criteria**				
	Abstraction Layer	Customization/ Reuse	Scalability	Autonomic Properties	Control Visibility
JADE [BPHT06]	*High, Generic*	*Yes*	*No*	*Self-CHOP*	*Yes/No*
TUNe [BHS+08]	*High, Domain-Specific*	*Yes*	*Yes*	*Self-CHO*	*No*
UNITY[CSW04] [TCW+04]	*High, Domain-Specific*	*Yes/No*	*No*	*Self-O*	*No*
AUTONOMIA [XSL+03] [YCHP05]	*Low, Domain-Specific*	*No*	*Yes*	*Self-CHOP*	*No*
RAINBOW[wC08] [wCcHG+04]	*High, Domain-Specific*	*Yes*	*No*	*Self-CHOP*	*Yes/No*
CEYLON [MDL10]	*High, Domain-Specific*	*Yes*	*Yes*	*Self-CHOP*	*No*
DiaSpec [CBCL11]	*High, Domain-Specific*	*No*	*Yes/No*	*Self-CHOP*	*Yes/No*

Table 2.1: Classification of Architecture-based Autonomic Solutions

The analysis of the classification of autonomic solutions provided on Table 2.1 shows many limitations of existing solutions for tackling the issue of engineering amenable autonomic solutions. This classification shows that many architecture-based autonomic solutions provide high level abstraction languages for implementing autonomic systems. However, these high level abstraction languages usually mask the visibility of the control architecture. Sometimes, even when the control architecture remains visible like in Jade or Rainbow, the implementation of its constituents are not transparent for developers. We also observe that the visibility of the control architecture can be reified at design time but disrupted at runtime like in DiaSpec. The customization or the reuse of components during the engineering process are diversely implemented. However, the type of mechanisms used for supporting the customization can have a strong impact on the scalability of the solution. For example in Rainbow, the choice of static control architecture with points of customization significantly impact the scalability of the control system, because it tends to promote a centralized control architecture. Finally, autonomic properties are diversely supported. Some solutions like Unity provide specialized autonomic managers that implement a specific autonomic property, and other a generic support for implementing all self-adaptive properties.

The comparison of existing software autonomic solutions presented in this chapter shows that none of existing solutions fully satisfies the engineering criteria of autonomic systems. In particular, a recurrent limitation is the lack of visibility of the control architecture. The lack of visibility of the control system is a serious threat for autonomic systems, because it makes them difficult to maintain and consequently increase their maintenance cost which the opposite of the target goal of the autonomic computing. We think that the following challenges need to be addressed for tackling the limitations of existent autonomic solutions:

- **Challenge 1:** Make feedback control loop explicit at design and at run time to enable their understanding and analysis.
- **Challenge 2:** Provide some verification techniques for evaluating the correctness of the control architecture in order to facilitate their maintenance.
- **Challenge 3:** Provide techniques that reduce the cost of engineering and maintaining autonomic systems.

The above mentioned challenges were discussed and acknowledged as relevant research avenues for autonomic computing in specialized international conferences [CdLG+][BSCG+09]. In the contribution part (cf. Part II), we present how these challenges are addressed within this thesis.

2.4 Summary

In this chapter, we have introduced autonomic computing and presented the objective of this research field. We have surveyed some existing autonomic software solutions, and com-

pared them according to defined criteria. From this comparison, we outlined some challenges that need to be addressed in order to improve the engineering of autonomic system. In particular, we underlined the visibility of the control architecture at design time and at run time as an essential requirement towards building amenable autonomic systems.

In the next chapter, we introduce the SALTY model which is a control oriented model that serves as a ground for this thesis contribution. The originality of the SALTY model is to enable the description of control systems where feedback loops are reified as first-class entities and represented with respect to the MAPE-K paradigm.

This page was intentionally left blank

Chapter 3

Salty Model

"Try to learn something about everything and everything about something."
– Thomas Henry Huxley

Contents

3.1 SALTY Structural Model . . . 34
3.1.1 Requirements . . . 34
3.1.2 Concepts . . . 35
3.2 SALTY Graphical Formalism . . . 37
3.3 SALTY DSL . . . 39
3.3.1 Control Elements . . . 39
3.3.2 Connectivity . . . 40
3.3.3 Non-functional Properties . . . 41
3.4 Summary . . . 42

In the previous Chapter (cf. Chapter 2), we have presented the state-of-the art of autonomic computing, and pointed out the limitations of existing works for building manageable autonomic systems. In this chapter, we introduce the SALTY model which is used as a basis for our proposal discussed later in the contribution part (cf. Part II) of this thesis. The SALTY model has been developed within the *ANR SALTY* project through which the current thesis was funded. The SALTY project aims at proposing a new step ahead regarding runtime self-adaptation of large scale distributed systems. In order to achieve this objective, the SALTY project relies on the usage of autonomic computing and control theory principles, for building an end-to-end Model-Driven approach [Ken02] using models for and at run-time, with representation of large-scale distributed system at different abstraction levels.

The SALTY Model fits into the vision of the SALTY project, by proposing control oriented concepts that enable the modelisation of *Self-Adaptive Systems*–ie, systems that are able

to adjust their behavior with regards to some defined goals in response to their perception of the environment and the system itself, and do that autonomously. The aim of the SALTY model is to ease the process of designing the architecture of self-adaptive systems. It allows an explicit definition of the semantics of a self-adaptive behavior without any major technological concerns. The idea is to focus on an *externalized adaptation*, that is to introduce self-adaptive behavior into already existing legacy systems and support different degrees of separation with respect to the target system. This is an alternative approach to the more traditional mechanisms, which allow these systems to detect and recover from errors, that are typically wired at *the level of code* where they are hard to change, reuse or analyze [GS02]. The use of the SALTY model combined with our proposal empower software engineers to turn existing systems into self-adaptive systems in time and cost effective manner.

Structure of the Chapter
This Chapter is organized as follows. Section 3.1 discusses the concepts of the SALTY model. Section 3.2 introduces the graphical formalism used to represent a control architecture using the SALTY model specification. In Section 3.3, we introduce the SALTY language, which provides a means for designing feedback control loops with the concepts of the SALTY model. Finally, we conclude this chapter with a summary.

3.1 SALTY Structural Model

The structure of the SALTY model is represented as an EMF Ecore model. The model is in a nutshell a graph of distributed adaptive elements with various profiles: sensor, effector and controller and associations between them. The model is defined using a meta- modeling approach (similar to the UML).

3.1.1 Requirements

In this section, we outline the main requirements for building self-adaptive systems presented in [CdLG+], and that have guided the specification of the SALTY model.

- Making the control loops explicit and exposing self-adaptive properties to allow the designer to reason about the system modeling support for control loops.
- Support for different types of control composition. Instances of feedback control loop (FCL) can also be created to control different aspects of other FCL on a control system architecture.
  - FCLs are to be instantiated to coordinate other loops at the same level, forming a hierarchy of loops.

- FCLs will aim at controlling control elements (monitors, etc.), thus being loops at the meta-level. More generally, any FCL element can be self-managed in the same way.
- FCL might have to control the behavior of several or all FCLs, also from the meta-level. For example, this will be used to enforce time constraint on the overall self-adaptive parts or any constraints on the features of the loops.

3.1.2 Concepts

This section summarizes the main design principles that were used in the creation of the structural model. The adaptation semantics, the definition of the feedback elements with their relevant inputs and outputs for data and control flows, is captured in a technologically agnostic model where each phase of the feedback control is made explicit and reified as an first-class element. The targeted system, the subject of the adaptation, is only represented through its touch-points: *sensors* for context observation (*sensor elements*), and *actuators* from which the adaptation is performed (*effector elements*). The core of the adaptation, the decision making process is encapsulated in the *controller element*. There is no assumption on which kind of controllers should be used in the system. The model also explicitly defines data and control flows in the architecture via links that express the dependencies among the respective elements.

A) Type and Instances

The presented model contains both the type definition as well as the instance definition. At the type level, the complete structure of each adaptive element is described. This involves definition of its *properties* and *features*, *data* and *control link* binding, *composite exports* and *ports imports*. At the instance level we define a tree of *AdaptiveElementInstance* following the features that were defined in their corresponding types together with *PropertyValue* for their property values.

B) Adaptive Elements

For a system to be self-manageable, it requires to be aware of itself. Therefore, besides having the ability to observe the target system and the running environment, the adaptive system also needs to know about its structure and behavior. For this, we use the *AdaptiveElementType* as the main conceptual element in the model. It represents an abstract entity in the system which the concrete behavior is defined by one of its subclasses. An adaptive element can, if needed, provide its own sensors and effectors in order to present information about its state or meta-data and to offer a way to adjust its behavior respectively. This reflective ability thus allows to make explicit loops and their elements, to compose different loops, but also to build a hierarchy of control on top of controllers as well as on top of any other adaptive element.

Data Providers - DataProvidingType. *DataProvidingType* represents elements that are part of the monitoring subsystem, elements that supply information about the underlying system into controllers so they can reason about the state of the target system and its environment. There are two main kinds: a *SensorType* and a *FilterType* that are through their data dependencies hierarchically organized in a form of a directed acyclic graphs. The leaf nodes in the hierarchy - *SensorType* - provides data of a defined *DataType* that are directly gathered from an external entity like various operating system resources, services calls, user preferences, etc. The other nodes that are not leaves - *FilterType* - are used to aggregate or in some other way process data that are coming from one or more connected sensors. They can be real data filters, stabilization mechanisms, converters, rules inference engines.

Data Processors - DataProcessingType. *DataProcessingType* represents elements that process data that comes from the *DataProvidingType*. There are two main kinds: a *FilterType* that has been described above and *ControllerType*. The latter represents the decision making part in the model. In general, its responsibility is to choose an appropriate action among the set of all permissible actions based on the observed state of the subjected system. There are many different kinds of controllers that can be used for the decision making process.

Effectors - EffectorType. *EffectorType* carries out the actual system modification and is orchestrated by one or more controllers. An effector encapsulates a set of operations Operation that can be used by a *ControllerType* via *ControlLinkType*. An operation is a named action that can take an arbitrary number of *OperationParameters* and whose purpose is to adjust the target system.

Links - LinkType. *LinkType* is responsible to deliver data from a data provider to a data processor in case of a *DataLinkType* or to invoke an operation requested by a controller on an effector in case of *ControlLinkType*. The data flow in the system originates in sensors and progresses through connected filters to terminate in controllers.

Ports - PortType. *Ports* represents the means to support distributed elements with remote communication. The remote ability is an important aspect of a system that allows it to spread its operation over multiple hosts possibly running in a distant places. However the interaction between elements in a distributed system need to be dealt with in ways that are intrinsically different from objects that interact in a single address space.

C) Structural Parts

The *StructuralPart* represents an abstract structural part of an AdaptiveElementType. There are two types of structural parts available: a *property*, and a *feature*. They correspond roughly to the concepts of an attribute (*EAttribute*) and a reference (*EReference*) from EMF. Each *AdaptiveElementType* can define one or more Properties whose values will be provided at the in-

stantiation point and kept in *AdaptiveElementInstance* using *PropertyValue*. A *feature* represents a reference to another adaptive element type.

D) Datatype Parameters

Datatype parameters introduce a generic datatype declaration that allows to create a generic reusable elements. Each *AdaptiveElementType* can declare zero or more *DataTypeParameters*. The following elements require definition of a datatype *DataTypeDefinition*, they all inherit from the *DataTypedElement*: *DataProvidingType*, *OperationParameter*, *Property*, *DataLinkType*, and *DataTypeArgument*.

E) Operation Selectors

In order to foster reuse, a generic operation selector *SelectorParameter* can be used at the places where an operation reference is needed. Besides just introducing a wildcard representing any operation name, additionally it has to specify the *datatype* of all its arguments (zero or more). The following elements require to define an *Operation*: *ControlLinkType* and *SelectorArgument*.

F) Composites

A control system, represented by the *ControlSystem* element, is composed of one or more *CompositeTypes*. A *composite* is an element that allows to hierarchically compose other elements and form a tree of adaptive elements. Any element defined inside a composite can be exported and thus made available to the outside.

G) Binding

A binding establishes a connection between a source *LinkType* and a target element depending on the concrete type of the link. The binding entity *Binding*, together with *LinkType* forms the data and control flow in the system.

3.2 SALTY Graphical Formalism

The SALTY model syntax enables developers to describe control system architectures. Figure 3.1 depicts the graphical notations that can be used by developers in order to design control system architectures using the salty specification. The Figure shows that the graphical syntax of SALTY is control oriented. That is, notations used for the graphical representation of a control architecture are all related to the notion of control. These notations enable to

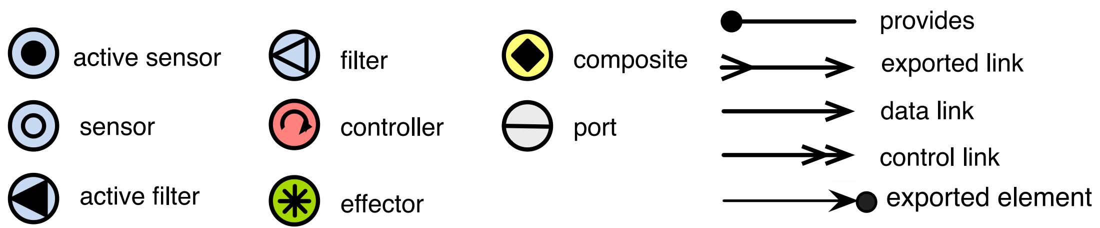

Figure 3.1: SALTY Graphical Notations

represent control elements– i.e. *sensors, filters, controllers, effectors*–, composition aspects–i.e. *composites, port*–, as well as the connectivity *–data link, control link, exported link, provided link* in the control architecture.

Illustrative example

To illustrate how to use the graphical notations of the SALTY model for designing a control

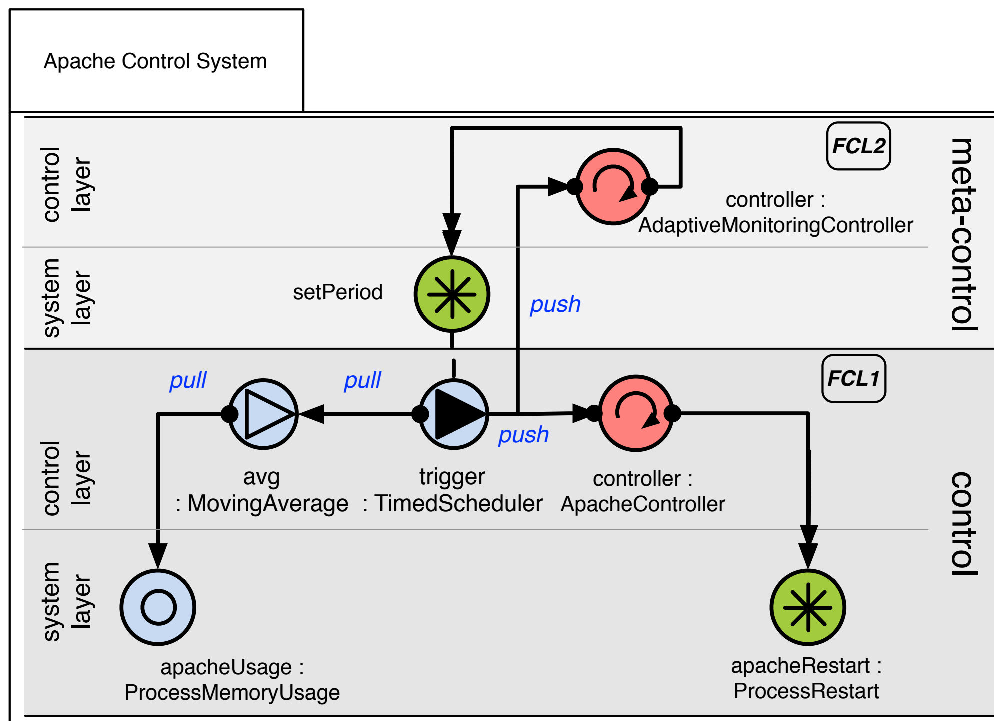

Figure 3.2: Example of Apache Control Architecture Representation with the SALTY Specification

architecture, let us consider the example of the *Apache usage memory* control architecture.

The *Apache usage memory control system*, monitors the memory usage of an Apache server. The behavioral policy for the *Apache control system* consists in restarting the Apache server when the usage memory exceeds *1 Gigabyte (GB)*. The *Apache control system* is able to adapt the frequency at which *memory usage* is monitored. Figure 3.2 illustrates how the *Apache control system* can be modeled using the SALTY graphical formalism.

Figure 3.2 shows that the *Apache control system* consist of two feedback control loops, *FCL1* and *FCL2*. The first control loop *FCL1* actually monitors the memory usage of the Apache server and decides or not to restart it. The decision logic of the feedback loop *FCL1* is contained in the controller, *ApacheController*. The second loop, *FCL2* monitors the frequency at which the memory usage is pulled from sensor *apacheUsage*. The monitoring frequency can be increased for example when the usage memory is getting close to 1GB. The feedback loop *FCL2* plays the role of a meta-controller, that is *FCL2* controls another feedback loop (*FCL1*).

The example depicted on Figure 3.2 shows that the semantic of the SALTY model is control oriented. This is relevant for developers of control systems, because they can explicitly describe the behavior of the control system with concepts close to their application domains.

3.3 SALTY DSL

There are basically two options for designing a control system with the SALTY specification. We can either use the Eclipse Modeling Framework (EMF) tool or the dedicated SALTY domain specific language (DSL). The first option is close to a graphical design, while the second option is a textual design. The SALTY DSL provides language constructions based on the Scala [pl] programming language to ease the design process of control systems. In this section, we are going to review some key language constructions of the SALTY DSL.In particular, we are going to highlight how the use of annotations help to tackle cross-cutting concerns. For example, the *CORONA* approach which is the main contribution of this thesis and presented on the next part (cf. Part II) of this document, leverage concerns like constraint analysis, code generation or non-functional properties through the annotation mechanism.

3.3.1 Control Elements

Basically, four types of control elements are generally used when designing control systems. That is, *sensors*, *effectors*, *filters* and *controllers* types. To illustrate how control elements are represented using the SALTY DSL, we are going to use the Apache control system presented in Figure 3.2 as a reference architecture. The listing 3.1 gives a snippet of the syntax for describing control elements in the SALTY DSL.

In the listing 3.1, we can notice that the definition of a control element consist in two main steps: The first step consists in the definition of the control element type and the second step consists in the definition of concrete instances for a given type of control element.

In the listing 3.1, the lines 2–20 depict the syntax for *ProcessMemoryUsage*, *ApacheController*, *MovingAverage* and *ProcessRestart* control element type. In particular, we can notice that the filter type *MovingAverage* has a property named *windowSize* and a datalink named *input*.

```
1     // control elements type definition
2     sensorType(name = "ProcessMemoryUsage", dtype = float)
3
4     filterType(name = "MovingAverage") init { implicit e =>
5       typeParameter(name = "T")
6       dtype = 'T
7       property(name = "windowSize", dtype = int32, defaultValue = int32("5"))
8       dataLinkFeature(name = "input", dtype = 'T, required = true, 'type' = 'DL)
9     }
10
11    controllerType(name = "ApacheController") init { implicit e =>
12     dataLinkFeature( name = "input", dtype = float,
13                      mode = push, required = true, 'type' = 'DL
14                    )
15     controlLinkFeature(  name = "action", operation = operation(int32),
16                          required = true, 'type' = 'CL
17                       )
18    }
19
20    effectorType(name = "ProcessRestart", operation = operation(int32))
21
22  // composite
23  compositeType(name = "Main", main = true) init { implicit e =>
24    //control element instances definition
25    //sensors
26     sensor(name = "apacheUsage", 'type' = 'ProcessMemoryUsage)
27     //filter
28     filter(name="avg", 'type' = 'MovingAverage)
29     //controller
30     controller(name = "controller", 'type' = 'ApacheController)
31     //effector
32     effector(name = "apacheRestart", 'type' = 'ProcessRestart)
33  }
```

Listing 3.1: Control Elements Design with the Salty DSL

Similarly, the lines 26–32 correspond to the definition of *apacheUsage, avg, controller* and *apacheRestart* instances. The definition of each instance indicates the type of the instance through the *type* attribute.

3.3.2 Connectivity

In the SALTY DSL, the connectivity between the elements of the control system are expressed through two types of bindings: That is, *controlBinding* and *dataBinding*. The main difference between these type of links is that *controlBinding* can exist only between controllers and

effectors. The Listing 3.2 gives an excerpt of the SALTY DSL syntax for defining data and control bindings. The lines 2–5 correspond to the definition of the dataBinding b1, and the lines 7–10 to the definition of the controlBinding b2.

```
1  //data binding definition
2    dataBinding(name = "b1",
3                source = sensorRef('apacheUsage),
4                target = dataLinkRef('avg, 'input)
5               )
6  //control binding definition
7    controlBinding(name = "b2",
8                   source = effectorRef('apacheRestart),
9                   target = controlLinkRef('controller)
10                 )
```

Listing 3.2: Connectivity Design with the Salty DSL

3.3.3 Non-functional Properties

Non-functional properties are essentially carried out in the SALTY DSL through the annotation mechanism. Annotations are used to integrate cross-cutting concerns during the design of the control architecture. They offer a flexible way to keep the core model lightweight while addressing other concerns that are not encapsulated in the core model.

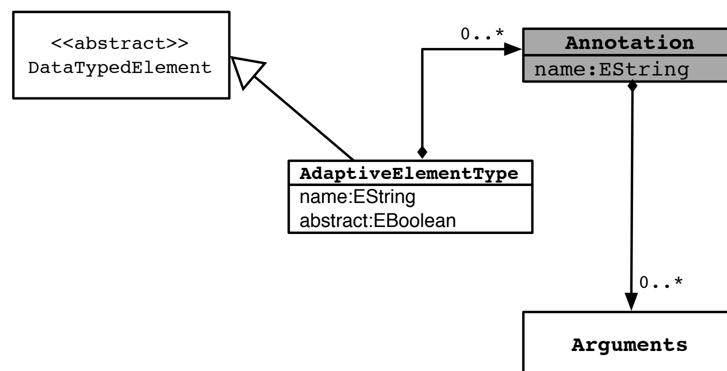

Figure 3.3: Annotation Class Diagram

Figure 3.3 depicts the *Annotation* class diagram. The figure shows that an *AdaptiveElement* can be decorated with annotations. Since, control elements and connection links in the SALTY model are derived from *AdaptiveElement*, the description of the latter can easily be enriched with annotations. The Listing 3.3 illustrates how a dataBinding can be decorated with an annotation.

The Listing 3.3 describes the @Stabilized annotation on the dataBinding b1. The use of the stabilized annotation will drive the code generator for generating a filter component of

type DeltaOperator between the *apacheUsage* sensor and the *avg* filter. The DeltaOperator filter is a stabilization algorithm which purpose is to regulate the variation of input value from the sensor *apacheUsage* to the filter *avg*. We elaborate on stabilization algorithms in the contribution part of this thesis.

```
1  //data binding definition
2    dataBinding(name = "b1",
3                 source = sensorRef('apacheUsage),
4                 target = dataLinkRef('avg, 'input)
5                ) init { implicit e=>
6                     //annotation definition
7                     annotation("Stabilized", DeltaOperator)
8                  }
```

Listing 3.3: Connectivity Design with the Salty DSL

3.4 Summary

In this chapter we have presented in details the concept provided by the SALTY model for developers of autonomic systems. We have seen that these concepts are domains specific and control oriented. The SALTY model enables developers to explicit control loop architecture when building autonomic systems.

However, having an explicit description of the control system architecture at design time is not enough. It is also important to keep the control system implementation explicit at run time. This chapter concludes the state-of-the-art part of this thesis. In the next chapters, we are going to introduce the main contribution of this thesis. In particular, we are going to explain how we keep explicit the runtime implementation of the control system for autonomic applications.

Part II

Contribution

Chapter 4

Contributions Overview

"Important thing in science is not so much to obtain new facts as to discover new ways of thinking about them."
– William Bragg

Contents

4.1 Challenges Revisited **46**
4.2 Goals Revisited **47**
4.3 CORONA in a Nutshell **48**
4.3.1 Development Process 48
4.3.2 CORONA Toolchain 50
4.3.3 CORONA Runtime 50
4.4 Summary **51**

In the first part of this thesis, we described the landscape of autonomic systems in the state-of-the-art. We presented how autonomic systems are engineered with existing approaches, and exposed their limitations. In particular, we pointed at the lack of visibility of the feedback control loops at runtime.

The second part of this thesis introduces the *CORONA* (*Control Oriented appROach for buildiNg Autonomic systems*) approach, which focuses on providing support for engineering amenable autonomic systems with explicit runtime adaptive capabilities. The originality of *CORONA* is that it enables a flexible management of autonomic systems, by reifying control loop elements as first-class entities at runtime.

In the first chapter of this second part, we recall the challenges of this work, then we present an overview of the our proposal. Our proposal is structured around the following points: the CORONA runtime and the toolchain. The former provides a runtime environment for the implementation of the business logic of control loops elements, and the later provides tools support for the code generation and the architecture analysis.

4.1 Challenges Revisited

The growing complexity of software systems today has fostered the demand for solutions able to cope with this complexity. In the state-art-of-art presented in Chapter 2, we acknowledged the relevancy of *autonomic systems* for tackling this issue. However, building *autonomic systems* rises a set of scientific challenges for software engineers and the research community. In our study, we gave a particular focus to architecture-based autonomic solutions.

Despite the great achievements of these last decades, we noticed that significant progress still need to be done for *autonomic systems* to keep their promises. Some of these requirements were largely discussed in specialized conferences and papers [ST09, MPS08, CdLG+, BSCG+09].

In particular, the following challenges were reported:

Design Challenge Although the *MAKE-K* architecture model of autonomic systems proposed by *IBM* is largely accepted by software engineers, alarmingly this model is usually used informally for discussing the logic behavior of autonomic systems, but rarely implemented in the final systems. The *MAPE-K* architecture defined essential features allowing to build autonomic systems, and namely *Monitoring, Analyze, Planning, Execution*, with all these features sharing a common *Knowledge* about the system state. Making the *MAPE-K* architecture an engineering reality, will foster the visibility of control loops and self-adaptive properties. That is, feedback loops that govern self-adaptation must become first-class entities when engineering autonomic systems. This will give a ground for the development of tools support for automatic verification and validation of autonomic systems.

Runtime Challenge In general, building autonomic systems is accomplished in two ways [LPH04]: *(i)* Extending programming languages or defining new adaptation primitives; *(ii)* Enabling dynamic adaptations by allowing reconfiguration capabilities of software entities (adding, removing, etc.). In practice, a combination of both solutions is required to build autonomic elements, and provide a runtime support for their interactions and management. Although numerous research efforts have investigated both solutions, there is still a lack of powerful framework that could help realize adaptation processes and instrument autonomic entities (sensors, effectors, etc.) in a systematic manner.

Mapping Challenge Given the new requirements that will introduce an architecture language that explicitly supports the description of control loops as first-class entities, it will be necessary to provide systematic methods for refining models in the language down to specific architecture runtime that support adaptation. That is because, depending on the runtime architecture support (component-based, aspect-oriented, product-line) there could be potentially a large gap in expressiveness between the runtime support and the architecture language. Therefore, a semi-automated process for mapping architecture language to the corresponding runtime support is required.

Validation and Verification Challenge Autonomic systems by virtue implement control loops each of which has their own objectives, but that must interact in order to deliver

a given service. Unintended interactions are potential source of conflicts and misbehavior of autonomic systems. Consequently, the development of autonomic systems requires techniques for validating effects of feedback loops interactions. In particular, identification of anti-patterns of control loops interactions is a relevant step toward the automatization of control loops validation, and the reduction of the cost entailed by their maintenance.

Reusability Challenge Functionalities implemented with an autonomic solution like sensors or decision-making, usually cannot be easily reused across others solutions. This requires developers to implement these functionalities over again when moving their applications from one solution to another one. That results in a lost of time and is error-prone. The complexity of autonomic systems and requirements with regards to their reliability call for generic and reusable solutions.

Scalability Challenge The need for enhancing software systems with self-adaptation capabilities principally stems in the complexity for managing them in a large-scale environment. That is because modern software systems are extensively distributed and heterogeneous. Consequently, an effective solution for building autonomic solution must address the decentralized architecture of the later.

4.2 Goals Revisited

To overcome the challenges mentioned above, we require a new global approach for software engineering of autonomic systems. The purpose of this thesis is therefore to provide software engineers with tools that support runtime self-adaptive capabilities as first class concerns. In the pursuit of this goal, we aim at achieving the following objectives for the proposed solution:

Application Domain Agnosticism An approach for building autonomic systems should aims to target a wide range of application domains and platforms. That is because, autonomic systems by virtue involve many stakeholders with different application domains. Therefore, an autonomic system approach should be generic enough to target a rich spectrum of domains. This will be a significant step to enable reutilization of existing components modules.

Transparency The approach should provide support for reasoning about self-adaptive capabilities as first class citizen at design and at runtime. That means that, it should allow software engineers to specify self-adaptation for distinct quality objectives, and integrate them to achieve self-adaptations across multiple objectives. The approach should also support the automation of verification tools for autonomic control loops. Finally, it should maintain a strong relationship between the expressiveness of the architecture language and the runtime capabilities to enable the evolution of systems.

Cost-effectiveness The solution should support low-cost efforts for engineering self-adaptive capabilities. This can be principally achieved through support for code generation,

verification automation and reutilization of implemented components. The automatic verification of autonomic systems architecture enables to implement conflict-free software, and provide a gain of time for developers as errors are detected at the early stage of the development process.

Modularity Since autonomic systems are usually deployed in a large-scale environment, the *modularity* of the solution is a key objective in order to have scalable self-adaptive systems. The modularity is also related to implementation of non-functional properties that must be as far as possible decoupled from the business logic of a given application.

In order to meet these objectives, we have developed a solution that reuse some good principles and techniques of software engineering like *Service oriented architectures* (*SOA*), and *model driven engineering* (*MDE*). In particular, our work support a clear separation between the adaptation logic and the business logic of the autonomic control loop.

4.3 CORONA in a Nutshell

4.3.1 Development Process

The *CORONA* approach enables the development of autonomic systems with an explicit description of feedbacks control loops architectures. *CORONA* aims at leveraging self-adaptive capabilities for software applications through the implementation of feedback loops.

The development of an autonomic manager with *CORONA* follows the traditional stages of a MDE approach. Figure 4.1 gives of an overview of the *CORONA* approach.

1. *Control architecture Design*: The *CORONA* engine toolkit takes as input feedback control loops architectural description based on *SALTY* DSL (Domain Specific Language), presented in Chapter 3. At this stage (cf. step ① in Figure 4.1), this initial architecture can be incrementally enriched through annotation-driven processes by invoking plugins services. Currently, *CORONA* includes one service based on *Constraint Satisfaction Problem* (CSP)[Apt03a] techniques for optimizing the distribution process. This service, called *Location Optimizer Service*, takes as input a control architecture instance model and the network topology model to compute an optimal distribution of control architecture modules for a given objective function.

2. *Architecture Verification*: Once the design phase is completed, and before the code generation of the implementation framework, the *CORONA* engine applies a set of heuristics and algorithms for detecting conflicts architectural patterns on the control loop architecture (cf. step ② in Figure 4.1). In case of failure, *CORONA* rises some warnings for the architect or the developer.

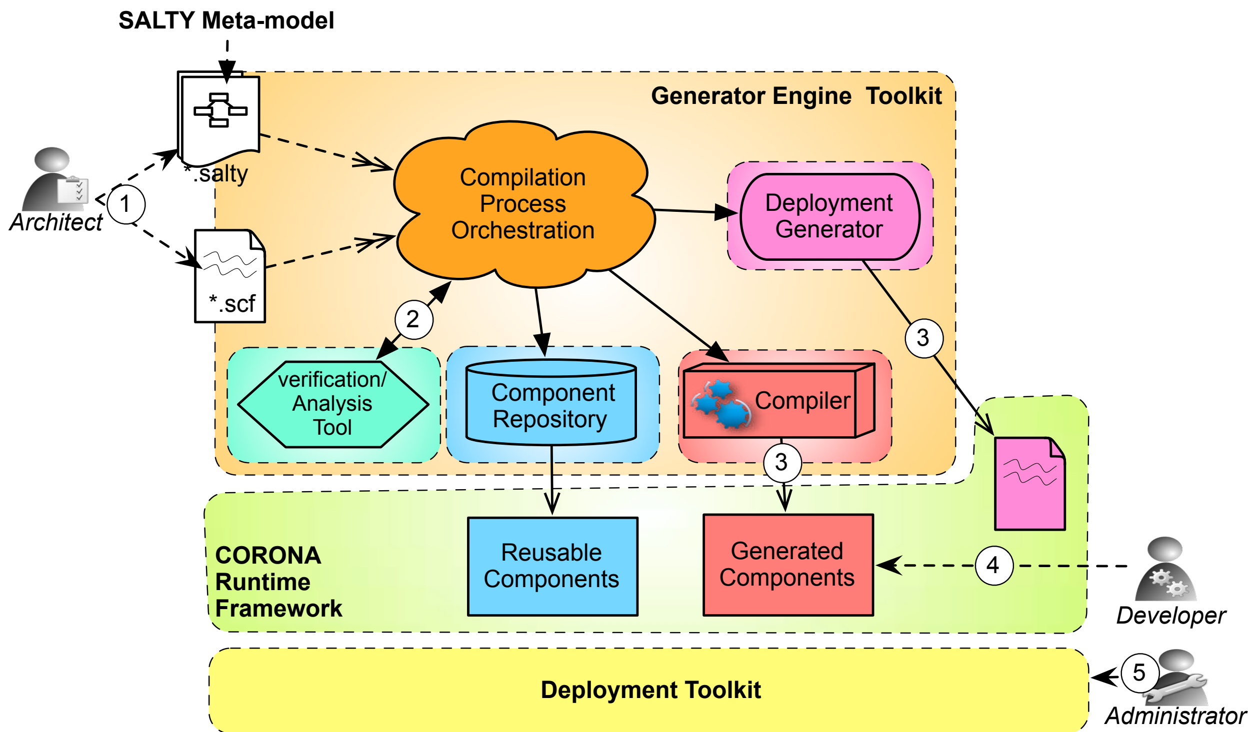

Figure 4.1: Overview Of the *CORONA* Development Process

3. *Code Generation*: At this stage, the final architecture of the autonomic control loop is forwarded to the *CORONA* compiler (cf. step ③ in Figure 4.1). The compiler generates a set of code artifacts corresponding to the architecture description (based on the Service Component Architecture - SCA [OAS07]), as well as code templates for the selected implementation technologies (*e.g.*, Java, Groovy, PHP,Web services).

4. *Implementation*: At this stage, the *developers* use the generated classes (cf. step ④ in Figure 4.1) to integrate the business logic implementation of autonomic loops nodes or to implement library modules corresponding to business concepts (specialized concepts).

5. *Deployment*: Finally, the autonomic system is deployed using deployment tools (cf. step ⑤ in Figure 4.1). In *CORONA*, the deployment phase consists in deploying components on different hosts using a dedicated tool called FDF [FDDM08] or generated *Fabric* [Fab] scripts.

The key features of the *CORONA* approach are the *toolchain* and the implementation *framework*. We discuss the role of each feature in the section below.

4.3.2 CORONA Toolchain

The CORONA *toolchain* maps the domain engineering to application engineering. It is responsible for several tasks such as the architectural checking, the mapping transformation between the SALTY DSL and the *CORONA* framework, and the code generation. On Figure 4.1 it is represented by the CORONA *engine toolkit*.

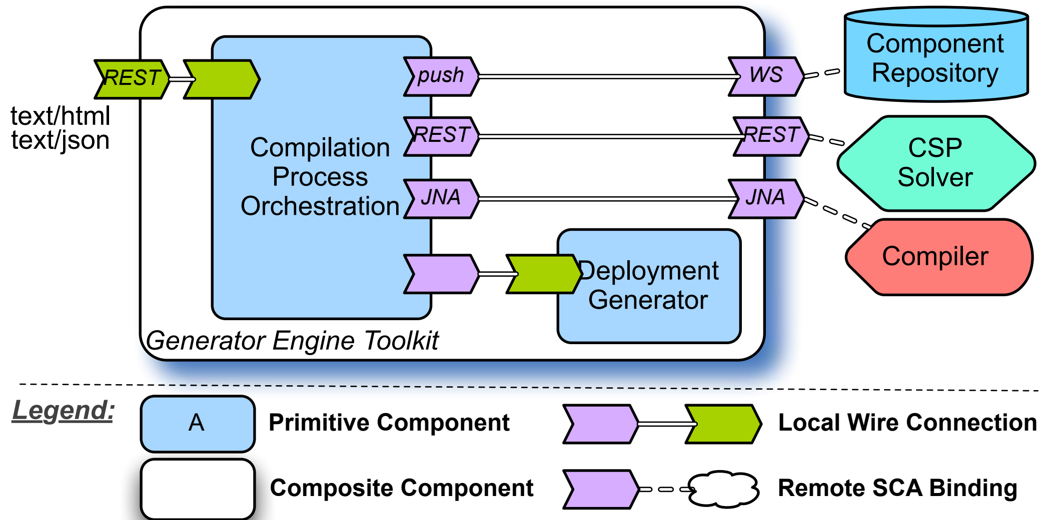

Figure 4.2: SCA Architecture of the *CORONA* Toolchain

The CORONA *toolchain* is built itself on a modular architecture based on service oriented architecture (*SCA*) principles. Figure 4.2 depicts the SCA architecture of the toolchain. The figure shows that the toolchain consists of independent distributed components that interact through SCA wire connection or bindings.

4.3.3 CORONA Runtime

CORONA framework provides the runtime support for feedback loops implementation. Currently, it is based on the *FraSCAti* [SMF+09] middleware, which is a runtime implementation of the SCA specification. Thanks to *FraSCAti*, the *CORONA* framework provides self-adaptive capabilities for the feedback loops components at runtime. The *CORONA* framework enhances the visibility of feedback control elements at runtime by providing specific components schemas for each type of control loop elements. This helps leverage issue related to autonomic loop runtime management, like *semantic carrying names* or *implicit access type*.

4.4 Summary

Modern software systems are getting more complex. This raises an important issue for their maintenance at a reasonable cost. Many works of the state-of-the art while, acknowledging the *MAKE-K* architectural paradigm for building autonomic systems do not provide explicit support for feedback loops elements neither at design nor at runtime. Feedback control loops that govern adaptation are usually hidden under abstraction layers of the implementation platforms. This result in difficulties for the development of automation tools for supporting the engineering autonomic systems.

In this chapter we have presented the objectives of this thesis, and gave a glimpse of the principles and basic concepts of the *CORONA* approach. We emphasized the originality of our approach which help coping the complexity of engineering autonomic systems by reifying feedback control loops as first-class entities at runtime. In the next chapter, we discuss the runtime architecture in the *CORONA* development process.

This page was intentionally left blank

Chapter 5

Runtime Architecture

"As far as the propositions of mathematics refer to reality they are not certain, and so far as they are certain, they do not refer to reality."
– Albert Einstein

Contents

5.1 Feedback Control Loops and Autonomic Systems 54
5.2 Runtime Component-based Feedback Control Loops 56
5.2.1 Control System Architecture 57
5.2.2 SCA Runtime Concepts 57
5.2.3 Feedback Control Loop Example 59
5.2.4 Anatomy of Control Elements 60
5.2.5 Interaction Model 66
5.3 Feedback Control Loop Customization 67
5.3.1 Properties of Feedback Control Loops 69
5.3.2 Stabilization of Decision-Making 72
5.4 Summary 79

In the foregoing chapter (Chapter 4), we presented scientific challenges that we aim at tackling in the context of this thesis. We also gave an insight of the contributions of this thesis in order to respond to these challenges. In particular, we emphasize one of the objective of this thesis which consists in providing software engineers with tools allowing them to implement autonomic systems where feedback control are reified as first class citizen at runtime.

In this Chapter, we go down in details to explain the *CORONA* approach for building amenable autonomic systems. In particular, we elaborate on the runtime architecture of the *feedback control loop* (FCL) elements. In the *CORONA* approach, self-adaptive capabilities

are realized through feedback control loops. They provide means to monitor activities of the underlying system, to take decisions upon made observations, and to trigger behavioral changes in the underlying system. *CORONA* allows the traceability of self-adaptive capabilities by reifying feedback control loops elements as first-class components at runtime.

The contribution presented in this chapter addresses two aspects of the feedback control loop architecture at runtime: *The implementation of feedback control loops elements* and *the customization of the feedback control loop itself*.

1. *Implementation of the feedback control loop elements.* Following the *MAPE-K* architecture pattern of the feedback control loop, we have defined the runtime implementation of each elements of the FCL. The architectural description is generic for each category of feedback control loop elements. However, the behavior of each control element can be refined to meet the requirements of a specific application. Keeping a generic architecture for FCL elements allow to instrument feedback control loop for verification purpose. We elaborate in details on feedback control loop verification on Chapter 6.

2. *Customization of the feedback control loop.* The *MAPE-K* pattern describes a static architecture of the feedback control loop. In practice, the implementation of feedback control loops requires to take into account cross-cutting concerns that are not captured by the *MAPE-K* pattern. Therefore, we have defined a meta-level on top of the *MAPE-K*, for the customization of feedback control loops that provide support for crosscutting preoccupations. In particular, our main contribution in this aspect is related to the stabilization of the decisions throughout the feedback control loop.

Structure of the Chapter The rest of this chapter is organized as follows: We start by presenting the architecture of a feedback control loop according to the *MAPE-K* pattern, and explain how autonomic behaviors can be built on it (cf. Section 5.1). Then, we continue by describing the component-based runtime architecture of the feedback control loop elements (cf. Section 5.2). In Section 5.3, we elaborate on the meta-customization of the feedback control loop. We give a particular focus to the stabilization of the control decision throughout the feedback control loop. Finally, we conclude this chapter (cf. Section 10.1).

5.1 Feedback Control Loops and Autonomic Systems

Feedback control loops consist of a set of components that collaborate together to maintain desired attributes for the controlled system. For example, an air conditioner maintains the temperature in a room to a desired value by sensing the air temperature and turning the heater or the cooler accordingly. The Reference *MAPE-K* model of a FCL presented in Chapter 2, consists of four essential activities of self-adaptation, namely *monitoring, analyzing, planning* and *executing*. However, IBM's *MAPE-K* reference model does not give any directives concerning the implementation of these activities. Therefore, some of these activities

can be performed manually by a human or automatically by a computing machine. This has lead to the emergence of two main paradigms related to the implementation of autonomic behaviors in software systems: That is *the close loop* and *open loop* paradigms [KBE99].

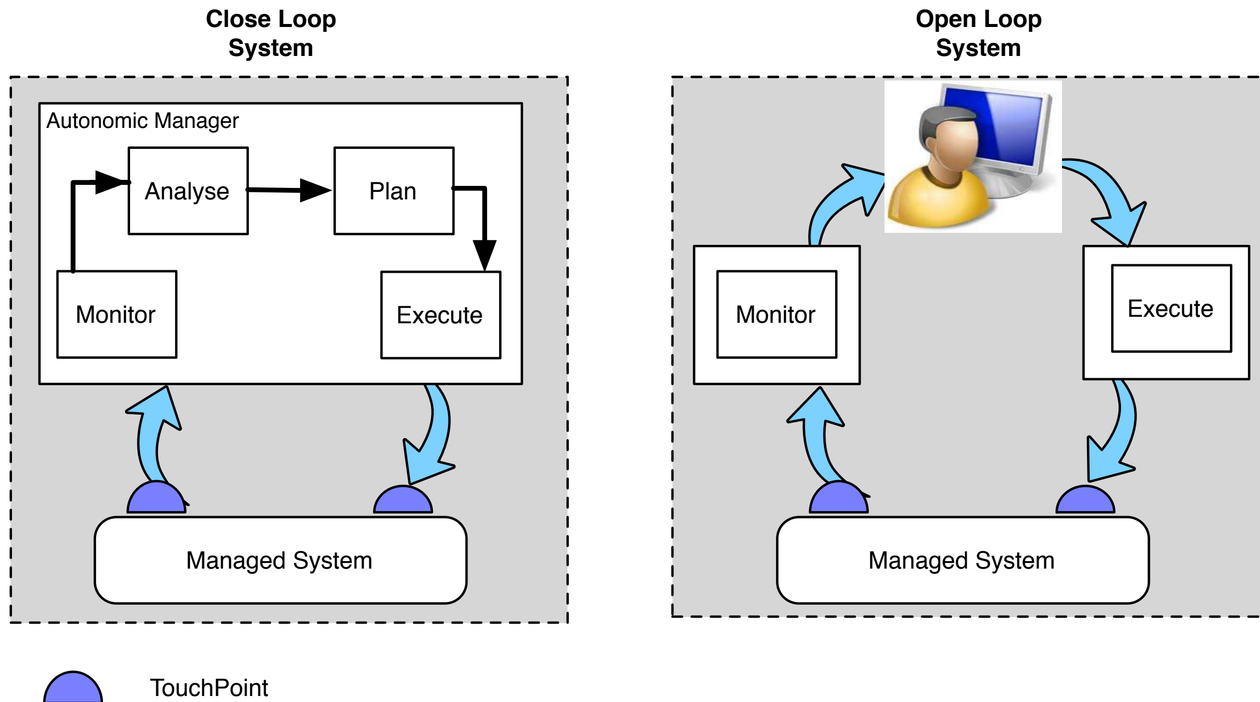

Figure 5.1: Close And Open Loop Paradigms

Figure 5.1 depicts the both paradigms. The *open loop* paradigm envisioned interactions with a human operator during the autonomic cycle of systems. In practice, activities devoted to the human operator are decision-making activities, and namely *analyzing* and *planning*. The *open loop* paradigm offers the advantage that the human operator is able understand and take smarter decisions upon conflicting policies regarding situations or events that were not and could not be anticipated before the system was actually running. On the other hand, the *close loop* paradigm supposes the automation of all activities of the feedback control loop. The latter has gained importance over the former, because it actually enables a system to evolve on its own without any external interventions, which is the closest behavior of an autonomic organisms in biology where the analogy of autonomic systems is drew from. However, as we discuss in perspectives of this dissertation, we think the future evolution of autonomic systems rely somewhere in between the both paradigms.

CORONA strives to combine both visions of the *close loop* and *open loop* paradigm. The *close loop* paradigm is supported in *CORONA* by enabling the automation of autonomic manager activities, and the *open loop* paradigm is leveraged through the visibility of autonomic managers architecture, which enables a human operator to interact transparently with the control system at runtime. Additionally, in the *CORONA* approach, the model of the *managed*

system is not required to build an autonomic system. The autonomic behavior is obtained by building the control system on top of the *managed system*. Interactions between the control system and the *managed system* is done through *touch points*, which are specific to each *managed system*. The *CORONA* approach enforces a separation between the implementation of the *managed system* and the control system. This enables feedback control loops of the control system to be visible at runtime as first class entities.

5.2 Runtime Component-based Feedback Control Loops

As we mentioned in Chapter 4, in many research areas feedback control loops are the central element for engineering autonomic systems. In particular, in control theory control engineers are provided with well-established tools and models for engineering autonomic controllers. Unfortunately, in software engineering despite some progress, feedback control loops that govern self-adaptation in autonomic systems remain hidden at design time and at runtime.

Self-adaptation enables software systems with the capacity to adapt their behavior depending on changes in their environment in order to maintain the desired objectives. Large-scale systems usually include many self-adaptive mechanisms in order to solve several classes of problems at different abstraction levels. The resulting complexity of the systems may lead to unexpected interactions with negative impact on the overall behavior of the system. Therefore, feedback control loops in autonomic systems must be explicitly described in order to enable control system checking for providing a reliable coordination of the self-adaptation.

CORONA enables to turn self-adaption in large scale systems by providing means to build feedback control loops on top of them. The particularity of the *CORONA* approach is that feedback control loops components are explicitly reified at runtime. In the *CORONA* approach software systems are empowered with autonomic capabilities by enhancing them with feedback control loops. Feedback control loops implementation is generated from architectural description based on the SALTY meta-model presented in Chapter 3. The *CORONA* approach relies on MDE techniques, which allows to target multiple implementation platforms (Web-services, OSGI, etc.) from the same model. However, in this thesis we focus on component-based implementation platform with respect to the SCA standard. The SCA platform offers the flexibility of SOA, and the reconfiguration properties of CBSE. In SCA, services are implemented as software components using various programming languages and technologies [SMF+09][OAS07].

The *CORONA* approach implements 3 principles in order to facilitate the process of building autonomic systems: *Separation of Concerns (SoC), Runtime reification of FCL architectural elements, and reuse of their runtime implementation*.

1. **Separation of Concerns**. Beside intrinsic business logic of each activities, feedback control loops must implement non-functional behaviors. To increase the readability

and the traceability of the control behavior implementation, the *CORONA* approach enforces the separation of concerns that must be implemented by the control system. This is made possible by decorating the control system model with annotations specific to each concern. The separation of concerns enables to have customizable feedback control loop rather than a static one where the business logic of the control is indistinguishably mixed with non functional concerns. We elaborate in details about feedback control customization in Section 5.3.

2. **Runtime reification**. *CORONA* ensures the mapping between the control system architectural description and its runtime component-based implementation in SCA. The SCA platform allows to represent feedback control loops and their activities as components compositions. This raises the visibility of the feedback control loop at runtime, and enable to reason on them for validation or verification purposes. An illustration of verifications that can be done on the control system architecture is presented in Chapter 6

3. **Reuse of implementation behavior**. One of the aim of *CORONA* is to reduce the cost entailed with the engineering of feedback control loops. In this perspective *CORONA* enforces the reuse of off-the-shelf components. From the architectural description of the feedback control loop, developers can specify new implementation type of the feedback control loop element, or use registered implementation type existing in the *CORONA* repository library. This reduces the burden of implementing control behavior logic.

5.2.1 Control System Architecture

5.2.2 SCA Runtime Concepts

The default target implementation platform of feedback control loops in the *CORONA* approach is SCA [OAS07]. *CORONA* uses the FraSCAti [SMR+12] runtime which is a well known implementation of the the SCA specification. Prior to present runtime anatomy of the feedback control loop using the *CORONA* approach we give an insight of SCA main concepts.

Figure 5.2 gives a simplified description of the main concepts and relationships between them in the SCA meta-model.

- **Component** is the basic element of business function in the SCA assembly, which are combined into complete business solutions by SCA composites. Some characteristics attributes of components are *services*, *references*, *properties*, and *component implementations*. *Properties* are attributes which are used to configure data values. *Component implementation* specifies the type of the implementation behavior for the component. SCA provides an extensibility point in the assembly model for supporting multiple

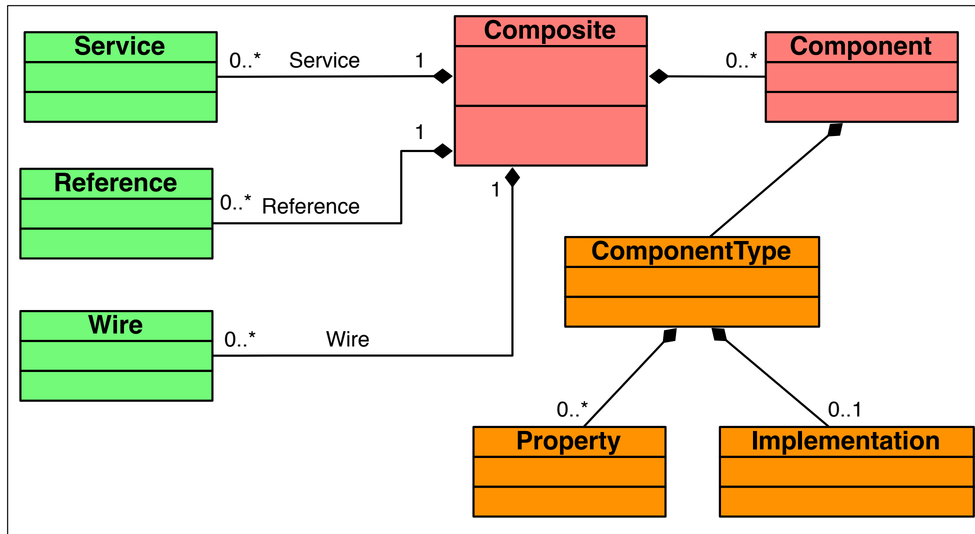

Figure 5.2: Basic concepts of the SCA Metamodel

implementation types as Java, BPEL or C++ implementation. Figure 5.3 depicts the graphical formalism of an SCA component.

- **Composite** is used to assemble SCA elements into logical groupings. It is the basic unit of composition within an SCA Domain. A *composite* contains set of *components*, *services*, *references* and *wires* for interconnecting all these entities. *Composites* can be used as component implementations in higher-level composites.

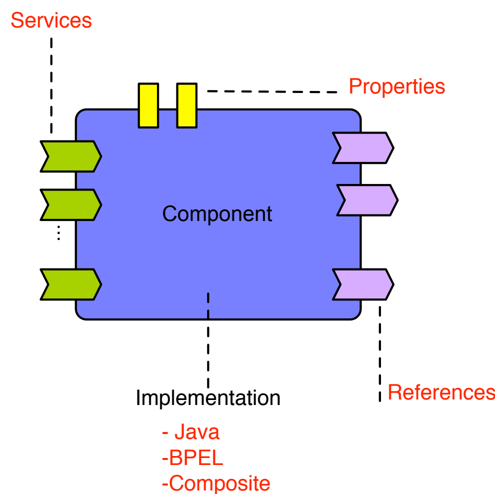

Figure 5.3: Graphical Representation of a Component in SCA

- **Service** represents an addressable interface of component implementation. *Services* are the entry points of to the component/composite business logic. A component can have zero or more service elements.

- **Reference** represents a requirement that a component has on a service provided by another component. Components can have zero or more reference elements.

- **Wire** is defined within a composite, and expresses the connection between component references and services.

5.2.3 Feedback Control Loop Example

To illustrate the anatomy of the feedback control loop runtime elements, we consider a feedback control loop for monitoring allocations of new resources for *Condor* self-adaptive cloud infrastructure. *Condor*[2] is a software system that creates a High-Throughput Computing (HTC) environment. Condor provides powerful resource management by match-making resource owners with resource consumers. The complete scenario of the *Condor* case study is presented in details in Chapter 7. In this scenario, we try to implement an autonomic system which is able to adapt resource allocation in the *Condor* environment depending on jobs workload and the price of resource leasing. One feedback control loop of the *Condor* case study monitors the rate of incoming job requests in order to deliver the target quality of service by ensuring the adequate allocation of resources available in the environment. Figure 5.4 depicts the architecture of the auto-scale feedback control loop using the SALTY specification.

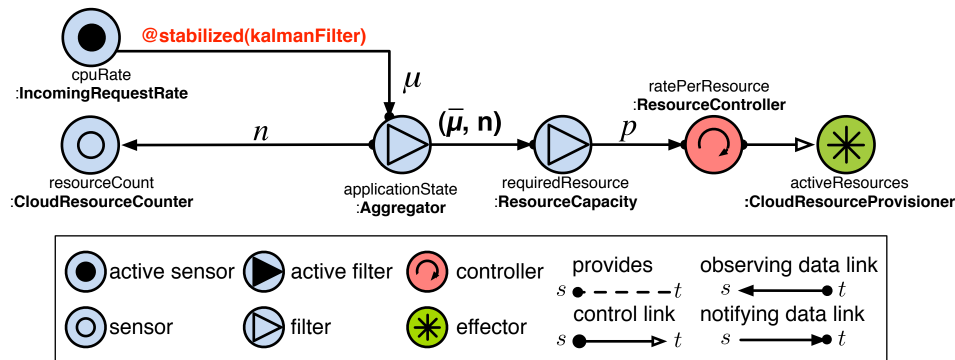

Figure 5.4: Auto-Scale Feedback Control Architecture

The feedback architecture depicted on Figure 5.4 is composed of 6 architectural elements that implement the MAPE-K feedback control loop activities.

[2]http://research.cs.wisc.edu/htcondor/

- **Monitoring**. This activity of the *MAPE-K* model is implemented by *cpuRate* and *resourceCount* sensor elements. The *cpuRate* sensor is an *active* sensor. This means that the sensor autonomously *pushes* information data (connections rate μ) to the target consumer, *applicationState*. Inversely, *resourceCount* is a passive sensor from which information data is *pulled* by *applicationState* filter. The connection rate μ measurement sway randomly between two computing cycle of *applicationState* filter. Since the accuracy of the latter measurement have a strong impact on the decision about resource allocation in the cloud infrastructure, we can apply a stabilization algorithm on that measurement to increase its accuracy. This is actually done with the annotation *@stabilized(kalmanFilter)* on the link between *cpuRate* sensor and *applicationState* filter on the feedback control loop architecture. The annotation *@stabilized*, drives *CORONA* for generating the runtime implementation of the "kalmanFilter" stabilization algorithm.

- **Analyzing**. This activity is implemented by the filters *applicationState* and *requiredResource*. The filter *applicationState* aggregates data from *cpuRate* (μ) and *resourceCount* (n) sensors. The filter *requiredResource* uses these data for calculating the resource capacity criteria, which quantifies the ratio between the number of connections and the number of allocated resources.

- **Planning**. This activity is implemented by the *ratePerResource* controller which decides to increase or decrease allocated resources. The decision for increasing or decreasing resources depends on the logic implemented by the *ratePerResource* controller. For example, for a simple *threshold policy*, new resources are allocated when we are below the threshold, and removed in the opposite case.

- **Execution**. This activity is implemented by the *activeResources* effector, which interacts directly with the cloud infrastructure to execute the decision of the controller. Typically, this consists in invoking native access method of the cloud infrastructure. That is, in the case of the auto-scale FCL, methods for resource provisioning.

5.2.4 Anatomy of Control Elements

This section presents the runtime anatomy of control elements that implement autonomic managers in *CORONA*. We exemplify the anatomy of control elements through the auto-scale feedback control architecture depicted on Figure 5.4.

The *CORONA* toolchain generates a set of Java classes and component descriptors for each adaptive element (*sensors, effectors, filters, controllers)* declared in the model. It also generates the component assembly and the interaction wires between components. Each Java class implementing an entity behavior is provided with a set of methods that encapsulates the default behavior of the current entity.

The *CORONA* toolchain reads the control system architecture model as the one depicted on Figure 5.4 in order to generate the runtime skeleton for the control architecture. Thanks

to the model-based generation of the source code, we leverage all the benefits of a MDE approach. Therefore, from the same architecture of the system we can target several implementation platforms depending on the target system requirements. However, in the context of this thesis, we put the focus to SCA/Java implementation platform. Thanks to the good properties that SCA exhibits in terms of *modularity*, it appears to be a good candidate for representing feedback control loop at runtime as first class citizen.

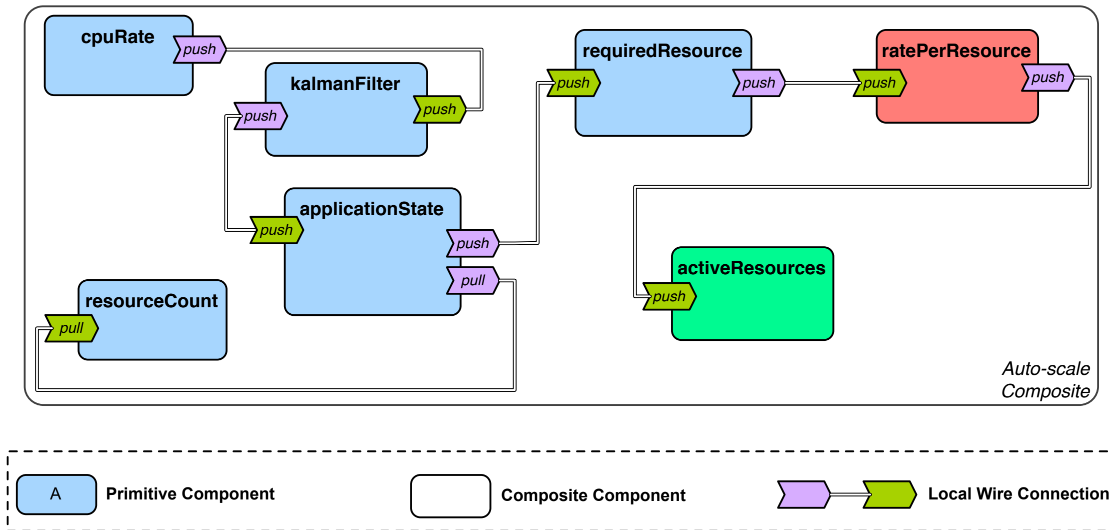

Figure 5.5: Generated Auto-Scale Feedback Control Loop Architecture in SCA

From the architectural description of the feedback control loop depicted in Figure 5.4, *CORONA* toolchain generates the runtime implementation of the auto-scale feedback control loop in SCA. Figure 5.5 shows the runtime architecture of the auto-scale control system generated by *CORONA*. We can notice that this architecture reifies the feedback control loop control elements as first class components. *Sensors*, *filters*, *controllers*, *effectors* are represented at runtime by components instances that implement their behavior. Hence, *CORONA* makes explicit at runtime the control system architecture.

Lets take a look at the concrete anatomy of the control elements generated by *CORONA* toolchain. As *CORONA* tends to be flexible, the structure of the generated elements are generic for the same type of control elements. That is *Sensors* types will share a common implementation architecture, as well as *filters*, *controllers*, or *effectors* types. In the rest of this dissertation, we will indifferently use the term *effectors* or *actuators* to refers to the same concept.

Sensors Artifacts

Sensors are one of the control element that compose the feedback control loop architecture. *CORONA* generates sensors runtime artifacts corresponding to each instance of sensor element that are depicted in the feedback control loop architecture. On Figure 5.5, *cpuRate* and *resourceCount* SCA components represent the generated sensor artifacts of the auto-scale

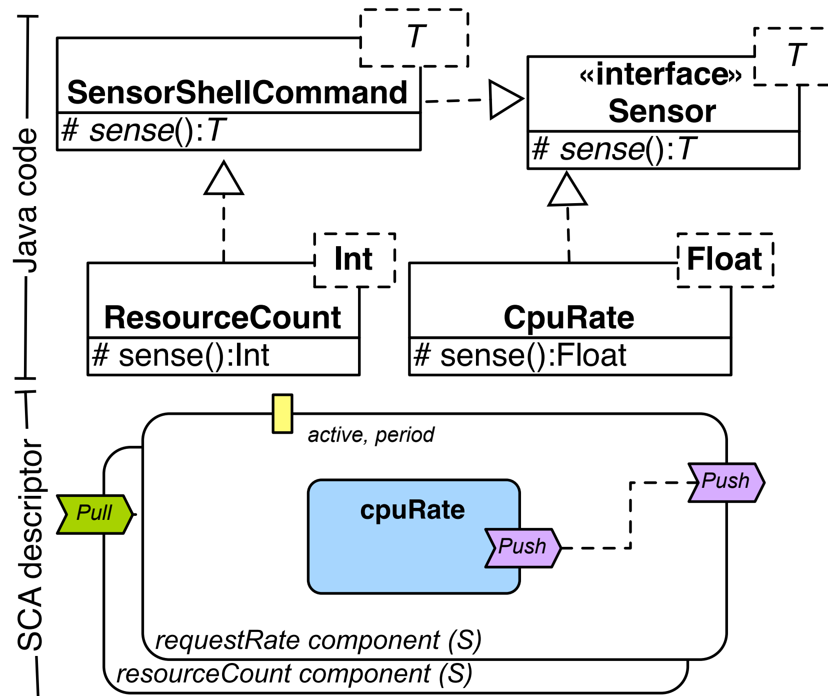

Figure 5.6: Generated Sensors Artifacts

feedback control loop (cf. Figure 5.4). Generated artifacts consists of *Java classes* and *SCA descriptors*. Figure 5.6 gives an overview of generated sensors artifacts.

Each Java class implementing an entity behavior is provided with a set of methods that encapsulates the default behavior of the current entity. For example, the method *<T> sense ( )* – where <T> is the type of the collected data– in a Java class implementing a Sensor interface, encapsulates the application logic of the Sensor Component. This is actually the placeholder for the code that gives access to the touchpoint of the *managed system*.

Having a dedicated set of methods for each entity type, significantly improve the readability of the generated code source. In order to implement the complete behavior of the Sensor Component, the developer will just have to provide an implementation of the *sense()* method. The same methodology is adopted for implementing the logic behavior of all the other types of entity of the control system architectural model. The implementation class of the sensor also contains methods that allow the interactions between components. That is the *Pull* and *Push* interfaces.

```
1 public class ResourceCountImpl<int>
2         extends SensorShellCommand<int> {
3
4         public int pull() {
```

```
5            String command ="";
6                  super.setCommand(command);
7                  return  super.sense();
8          }
9 }
```

Listing 5.1: Generated ResourceCount Sensor Implementation Class

The Listing 5.1 gives an excerpt of the generated code for *ResourceCount* sensor. It shows that *ResourceCountImpl* inherits from an existing type *SensorShellCommand<int>* which is provided by the *CORONA* toolchain. This demonstrates the aspect of components reuse in the toolchain. The *pull ()* method returns the result of the execution of the *sense()* method defined in *SensorShellCommand<int>* class.

Filters Artifacts

Likewise sensors, filters are also control elements of the control system architecture. *CORONA* generates filters runtime artifacts corresponding to each instance of filter control elements that are depicted in the feedback control loop architecture. For example, on Figure 5.5 *appplicationState* filter component represents the generated runtime artifact of the filter control element of the same name depicted on the auto-scale feedback control loop architecture (c.f Figure 5.4). In addition, we can notice that another filter artifact (*kalmanFilter*) that is not represented as a control element in the auto-scale FCL architecture is generated. The latter was specified in the control architecture model (cf. Figure 5.4) through the annotation *@stabilized(kalmanFilter)*.

```
1<composite
2xmlns:frascati="http://frascati.ow2.org/xmlns/sca/1.1"
3xmlns:gt="cgenerated"
4...
5<component constrainingType="gt:ApplicationStateFloatInt32"
 name="applicationStateFilter" >
6 <implementation.java
7        class="..impl.ApplicationStateFilterImpl" />
8</component>
9
10<service name="applicationStateFilter_srv1"
11promote="applicationStateFilter/applicationStateFloatInt32_srv1"
12/>
13
14<reference name="applicationStateFilter_ref1"
15promote="applicationStateFilter/applicationStateFloatInt32_ref2"
16/>
17
18<reference name="applicationStateFilter_ref2"
19promote="applicationStateFilter/applicationStateFloatInt32_ref1"
20/>
21</composite>
```

Listing 5.2: Generated SCA description of ApplicationState Component

Filters and *controllers* are part of the decision process that occurred within the feedback control loop. In auto-scale architecture example, the *kalmanFilter* component plays the role of stabilization mechanism. Stabilization mechanisms [NRS09] impact the quality (e.g., accuracy, up-to-dateness) of information processed through the FCL. The quality of information in the *CORONA* approach is addressed as a crosscutting concerns. The separation of concerns in the *CORONA* approach offers the possibility to customize the feedback control loop architecture while keeping explicit its control behavior. These aspects of the FCL customization which are one of the contribution of this thesis is discussed in details in Section 5.3.

Generated filter artifacts consist of Java class implementation and SCA assembly description. Listings 5.2 and 5.3 gives a snippet of the generated SCA description for the component *applicationState* and the Java class implementing its behavior.

```
1 @Scope("COMPOSITE")
2 @Service(interfaces={ PushFloatItfc.class })
3 public class ApplicationStateFilterImpl  implements
  PushFloatItfc {
4
5 @Reference(name="applicationStateFilter_ref1")
6 private PullInt32Itfc applicationStateFilter_ref1;
7
8 @Reference(name="applicationStateFilter_ref2")
9 private PushApplicationStateDataTypeItfc
10                   applicationStateFilter_ref2;
11
12 public void filter (float data , PullInt32Itfc data2) {
13   //TODO: add your code here
14 }
15
16 public void push (float data) {
17  //TODO: add your code here
18  return;
19 }
20
21 }
```

Listing 5.3: Generated ApplicationState Filter Implementation Class

Listing 5.2 describes that the component *applicationStateFilter* consists of two SCA references (lines 14,18), and a SCA service (line 10). Line 7 of this listing indicates that the Java implementation of the component behavior is specified in the Java class *ApplicationState-FilterImpl*.

Listing 5.3 describes *ApplicationStateFilterImpl* Java implementation. We notice that this class has a dedicated method *filter ()* which should contains the code realizing the control

logic of the component. The method takes has input two parameters corresponding to incoming data from sensors *resourceCount* and *cpuRate*.

Controllers Artifacts

Controllers are another type of control elements that can compose a feedback control loop architecture. *CORONA* generates controllers runtime artifacts corresponding to each instance of controllers control elements that are depicted in the FCL architecture. For example, the auto-scale feedback control loop architecture consists of one controller entity: *ratePerResource*. This entity implements the logic of deciding upon the allocation or the deallocation of cloud resources. *CORONA* generates the runtime artifacts for this entity as illustrated on Figure 5.5. Controller runtime artifact consists of an SCA component description file and a Java implementation class. Listing 5.4 gives an excerpt of the generated Java implementation class *RatePerResourceImpl* for the controller *ratePerResource*.

```
1 @Scope("COMPOSITE")
2 @Service(interfaces={PushFloatItfc.class })
3 public class RatePerResourceImpl  implements  PushFloatItfc {
4
5 @Reference(name="RatePerResourceController_ref1")
6 private PullStringItfc applicationStateFilter_ref1;
7
8
9 public void compute(float data){
10   //TODO: add your code here
11 }
12
13 public void push (float data) {
14  //TODO: add your code here
15  return;
16 }
17
18 }
```

Listing 5.4: Generated RatePerResource Controller Implementation Class

The decision logic implemented by the controllers in the feedback control loop can be more complex to be hold within a simple Java class. In some cases, in order to achieve a decision controllers must implement complex models with learning mechanisms like *Markovian Processus* [SAH07], developed by a third party. Using the *CORONA* approach the Java class implementing controller component can execute a script allowing to call that model.

Actuators Artifacts

Actuators are a type of control elements that can compose a FCL architecture. *CORONA* generates actuators runtime artifacts corresponding to each instance of actuators control elements that are depicted in the feedback control loop architecture. For example, the auto-scale

architecture consists of one *actuator–activeResources–* as depicted on Figure 5.5. Similarly to *sensors*, *actuators* interact directly with the *managed system* through *touch-points*. Execution of the decision from controllers can be done by running specific scripts or using specifics application programming interfaces (API) provided by the *managed system*. Generated artifacts for *actuators* consist of SCA component description files and a Java implementation classes. Listing 5.5 gives an overview of the generated Java implementation class for *activeResources* actuator.

```
1 @Scope("COMPOSITE")
2 @Service(interfaces={PushFloatItfc.class })
3 public class ActiveResourcesImpl  implements  PushFloatItfc {
4
5  public void execute(String resource){
6   //TODO: add your code here
7 }
8
9  public void push (String data) {
10  //TODO: add your code here
11  return;
12 }
13 }
```

Listing 5.5: Generated ActiveResources Actuator Implementation Class

Listing 5.5 shows that the *ActiveResourcesImpl* Java class has a dedicated method execute() (line 5). This method must contained the code for accessing the *managed system*.

5.2.5 Interaction Model

So far we have put the focus on how the control elements were implemented at runtime. In this section we discuss the interaction model between the generated components. The interaction model between control elements of the feedback control loop inherits from the architectural description model. The communications between components is essentially data-driven. We essentially distinguish three modes of interaction: The notification, the observation and the multimodal mode.

Notification Mode
This interaction mode consist of sending notifications from a component to all connected components. It is also known as the *Push* mode. The *notification* mode is the characteristic of *active elements* like the sensor *cpuRate* of the auto-scale architecture (cf. Figure 5.4).

Observation Mode
In the *observation mode*, information data of a given control elements can be retrieved by another elements interested in that information. The *observation mode* is also known as the

Pull mode and is the characteristics of *passive elements* like the sensor *resourceCount* of the auto-scale architecture (cf. Figure 5.4).

```
1 // pull interface
2 public interface Pull<P> {
3         public P pull();
4 }
5 // push interface
6 public interface Push <T> {
7         public void push(T pushData);
8 }
```

Listing 5.6: Pull and Push Generic Interfaces

In SCA, communication between components is realized through *wires* or *bindings*. *Wires* are used to indicate local communications between components within composites while *bindings* characterized remote communication between composites. *Wires* or *bindings* in SCA establish connections between references and provided services which are defined through Java interfaces. In the *CORONA* approach the *notification mode* is implemented at runtime through *Push* interfaces (cf. Listing 5.6 line 6), and the observation mode through *Pull* interfaces (cf. Listing 5.6 line 2).

Multimodal Mode
The multimodal mode consist of the combination of the *notification* and the *observation* mode for interacting between the control elements. In the *CORONA* approach with the exception of *sensors* and *effectors*, control elements can implement simultaneously *Pull* and *Push* interfaces.

In this section, we have explained how the control system architecture is reified at run time in *CORONA*. In particular, we have showed that control elements that compose the feedback control loop architecture are implemented as SCA components at run time. So far then, we have essentially addressed functional aspects of the feedback control loop implementation in *CORONA*. The next section will rather address the integration of non-functional properties of the feedback control loop implementation. In particular, it focuses on the customization of the control architecture with stabilization mechanisms.

5.3 Feedback Control Loop Customization

If it is advisable to keep a distinguishable structure of the FCL at runtime, it is equally important to pay attention on how its constituent parts are implemented. Otherwise, the implementation of FCL elements can quickly result in a black box modules difficult to maintain, and that rather play the role of logical than functional entities [MPS08]. In the *CORONA* approach we use a good principle of software engineering that is the separation of concerns (*SoC*)[KM05] for integrating various concerns in the FCL architecture. The feedback control

loop architecture can be enriched with these concerns through the mechanism of annotations. This approach enforces the representation of control elements as first class citizen in the feedback control loop architecture while enabling the customization of the latter. The customization of the feedback control loop architecture result in an *adaptive* control system architecture rather than a *static* one.

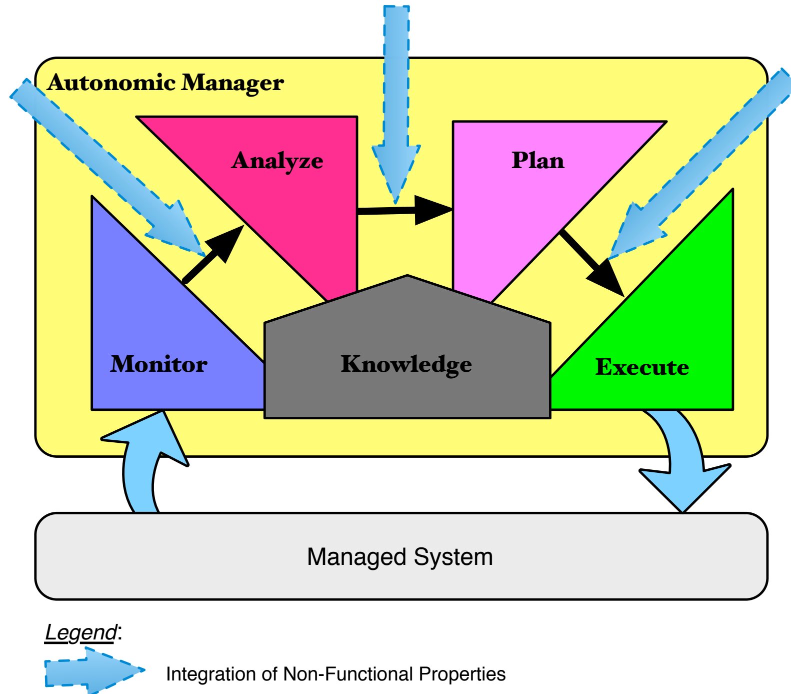

Figure 5.7: Customizable MAPE-K Architecture Model

Figure 5.7 illustrates our vision of adaptive FCL model. This figure depicts the MAKE-K *autonomic manager* model customized with non-functional properties. The integration of non-functional properties is represented on Figure 5.7 by the arrows with dashed lines. On the Figure 5.7, we can notice that non-functional properties are integrated transversally to the MAKE-K architecture model. This enables a flexible customization of the autonomic manager.

In the *CORONA* approach the customization of the autonomic manager is implemented through annotations. *CORONA* provides a bunch of annotations that enable to customize the behavior of autonomic managers. Several non-functional properties can be used to customize autonomic managers. For example, the security of the communications between distributed control elements, or the quality of the information processed throughout the feedback control loop. In particular, the latter has a huge impact on the nature of the control decision. One dimension of the quality of information (QoI) is the *up-to-dateness* [BS03]. *Up-to-dateness* informs about the age of the information. When decisions are about to be made for triggering an adaptation, *up-to-dateness* criteria can be a crucial factor for an effi-

cient *decision-making*. That is because, if a decision to trigger an adaptation is taken on the basis of an *old* information, it may not be pertinent for the system.

The quality of information has an impact on the decision made at the level of the FCL, and consequently on the relevancy of the triggered adaptations. In our approach, the quality of information in the feedback control loop is addressed through *stabilization mechanisms*. *Stabilization mechanisms* modify data acquisition processes. They aim at reducing the impact of data variability on the decision-making. The auto-scale architecture presented in Figure 5.4 depicts how the feedback control loop architecture can be customized with stabilization concerns in order to improve the decision-making.

5.3.1 Properties of Feedback Control Loops

The main objective of a feedback control loop consists in the regulation of a managed system, by making decisions for adapting the latter on the basis of observations of its states. For $s \in S$ the states of the managed system, and $e \in E(s)$ an action among the set of admissible actions, the function $\phi : S \rightarrow E$ captures the set of the decisions for the feedback control loop. These decisions are implemented through the *analyzing* and *planning* phases of the feedback control loop also known as decision-making phases. Consequently, one of the crucial task when implementing feedback control loop resides in the implementation of the decision policy.

In order to assess the behavior of a feedback control loop, it is important to define criteria which will serve as grounds for this assessment. In autonomic computing, to the best of our knowledge, there is no consensus about the definition of such criteria. However, in many other research areas where feedback control loops are largely used, sets of tools and property for evaluating feedback control loops exist. For example, the emphasis in control theory is on developing components and control algorithms such that the resulting system achieves the control objectives. Control engineers evaluate feedback systems through a set of four properties: *Stability*, *Accuracy*, *Settling time*, *Overshoot*, known as SASO properties. In the following sections we explain how SASO properties are used to characterize a control system. In particular, we put a special focus on the *Stability* property which is the single aspect of SASO properties that we have investigated in this dissertation.

Figure 5.8 described a block diagram of a feedback control in control engineering. The *reference input* is the control objective of the feedback block. The *controller* adjusts the setting of control input to the target system so that its measured output is closest to the reference input. *Transducer* represents some transformations that can be made on the measured output in order to be directly processed by the controller.

Stability characterizes a control system for which for any bounded input, the output is also bounded. In practice it corresponds to system without large oscillations of the output signal. The *accuracy* measures the convergence of the control system against the reference input. The *settling time* measures the time of convergence of the control system to its steady-state. Finally, *overshoot* measures whether the control system goes over a threshold reference capacity.

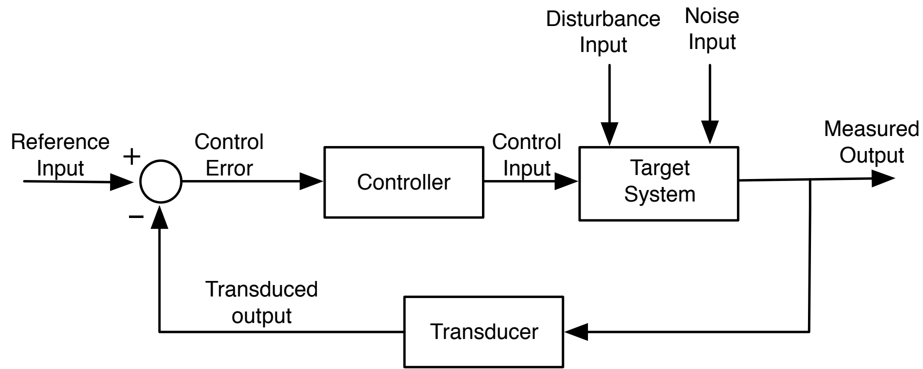

Figure 5.8: Feedback Block Diagram in Control Engineering

Autonomic computing and the control theory have in common feedback control loops as subject of study. Therefore, many researchers have tried to apply with more or less success some techniques of control theory in autonomic computing. One of the fundamental premise in this direction are the works of Hellerstein and al. [HDPT04]. In this thesis, we explore (cf. Section 5.3.2) the relationship between control theory and autonomic computing through the *stability* property of the feedback control loop. More precisely, we investigate how stable control systems can be engineered thanks to the use of stabilization mechanisms.

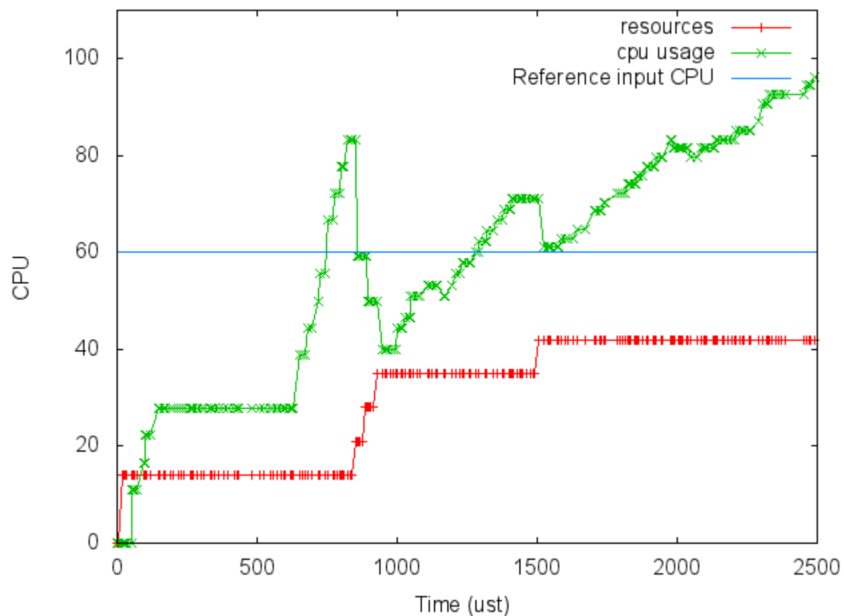

Figure 5.9: Unstable Auto-Scale Feedback Control System

Before going into details and explaining our contribution on the stabilization of control systems, let us illustrate why it can be important to have a stable control system. For that purpose, let us take a look at the plots on Figure 5.9. Figure 5.9 gives an example of an unstable control system. This control system corresponds to the architecture of the auto-scale feedback control without *kalmanFilter* component, and presented in Figure 5.5. The solid line without markers on Figure 5.9 represents the *cpu input reference* for the control system. The solid line with cross markers shows measured cpu values, and the solid line with vertical line markers shows the variation of resources. The instability of the auto-scale control system is characterized by the fact that measured values of the *cpu usage* oscillate around the reference input. From the control theory perspective, this means that the controller of the auto-scale system is not properly designed. In order words, the implementation logic of the decision-making for the auto-scale feedback control is not efficient.

The decision-making in the auto-scale FCL is captured within filters *applicationState*, *requiredResource* and the controller *ratePerResource*. The filter *applicationState* aggregates data (cpu rate, number of resources) from sensors and forward them to the filter *requiredResource* which tries to evaluate resource capacities for the given cpu load. The requiredResource filter, then forward information to the controller. The decision logic implemented by the controller is described by the algorithm 1.

Algorithm 1 ratePerResource Controller Implementation Logic

Require: $cpuRate, cpuReferenceInput, resourceCapacity$
Ensure: cpu usage in the system is less than the cpu reference input
  if $cpuRate \geq cpuReferenceInput$ **then**
    if $resourceCapacity > 0$ **then**
      Allocate New Resources
    end if
  end if

According to algorithm 1, *ratePerResource* controller monitors the value of the *cpu usage* and allocates new resources when this value is equal or above the *cpu reference* input value. If we desired to have a *stable* control system, we have basically two way of going from here: The first way consist of using a control decision model like the Markovian Decision Processes (MDP) for implementing the controller logic. However, the implementation of such models is complex and many software engineers can barely understand and build effective control decision models. The second way consist in tuning in a ad-hoc manner the logic behavior of the controller. The issue with the ad-hoc approach is that the code implementing the controller logic can grow quickly, and becomes difficult to maintain. The *CORONA* approach makes a tradeoff between the two aforementioned methodologies. In *CORONA* a stable control behavior can be achieved through stabilization mechanisms. *Stabilizations mechanisms* in *CORONA* consist of algorithms (including some models). Stabilization mechanisms impact the quality of the information processed within the feedback control loop, and consequently the decision based on these information. For example, in the case of the auto-scale FCL, the use of a learning stabilization mechanism like *kalmanFilter* can help to anticipate the workload of the cpu, and allocate resources before the reference threshold is

exceeded.

CORONA facilitates the integration of stabilization mechanisms in the control architecture, by providing support for a specialized annotation type– @stabilized("name of algorithm"). When the algorithm is referenced in the *CORONA* library, developers can benefit from it. In the next section we introduce some stabilization mechanisms and their characterization. This characterization helps developers to choose algorithms that meet their needs.

5.3.2 Stabilization of Decision-Making

Stability is one of the required property of a control system in control engineering. Control engineers are provided with set of tools and algorithms that drive them in order to implement a *stable* control system. For example, the calculation of the *transfer function* or the *pole*[HDPT04] of the control system helps them to build controllers with desired properties. Similarly, in *CORONA* software engineers can enhance autonomic feedback control loop with stabilization mechanisms in order to reach a *stable* decision-making.

This section is organized as follows: we first introduce a classification of stabilization algorithms, then a characterization of these algorithms in order to guide developers for choosing algorithms that meet their needs. After that, we present some emerging properties of stabilization algorithms that result from their composition. Finally, we illustrate the impact of stabilization algorithms in order to implement *stable* decision-making.

Classification of Stabilization Mechanisms

We based our classification of stabilization mechanisms on the type of stabilization strategies that they implement. Stabilization mechanisms are addressed in many works of pervasive computing related to context reasoning or context fusion of information [dRRM06, SAH07, Dar07].

Most of the proposals concerning the stabilization algorithms or techniques can be categorized into five groups:

Filtering techniques. These techniques focus on data-filtering using statistic or parametric-based algorithms. For example, in *MoCoA* [dRRM06], which is a framework for the management of network applications, data-filtering is achieved on the basis of geographical or temporal constraints.

Threshold techniques. For algorithms of this group, stabilization is realized by checking the system state with regards to threshold values. The *heartbeat* (HB) algorithm described in [TC04]—used for the stabilization of network connectivity failure in a mobile environment—is an example of a threshold-based technique.

Refresh techniques. The principle of these techniques is to update the system with new contextual information only when certain conditions are fulfilled. Usually these conditions are expressed in the form of time-based constraints or events/actions constraints.

Probabilistic schemas. For algorithms that belong to this group, stabilization is reached by inferring the system state on the basis of probabilities and previous states of the system. Examples of this category include *Kalman Filter* [PZLB05] and *Hidden Markov Model* [SAH07].

Uncategorized. This last group encloses all the algorithms that do not fit in the previous categories, because they use ad hoc methods to handle the stabilization of the system.

From this classification of stabilization algorithms, we are going to provide a characterization of these algorithms for helping developers to choose algorithms that best fit their needs when engineering autonomic systems.

Characterization of Stabilization Mechanisms

The characterization of stabilization mechanisms can be based on several criteria like the *algorithmic accuracy*, or the *algorithmic complexity*. Depending on the situations, these criteria may have different relevancy for the control system. For example, in some situations the accuracy of the results can be preferred over the speed of producing these results. We retained the two following criteria for characterizing stabilization algorithms: the *algorithmic data scope* and the *algorithmic responsiveness*. Our choice for these criteria is motivated by the fact that they are generic enough to be easily derived from most of stabilization algorithms.

Algorithmic Data Scope
The *algorithmic data scope* criterion informs about data input type (nominal, ordinal, numeric) for an algorithm. The above mentioned group of algorithms have various *data scope*. The *data scope* criterion is based on the data type categories proposed by Mayrhofer and al. [MRF03]. Authors suggest that primitive types of contextual information can be grouped in four categories:

- *Nominal data* (qualitative) includes values in a dataset on which no order relationship has been or can be defined. An example of *nominal data* are binary features for which values are defined in the set $S = \{0, 1\}$.
- *Ordinal data* (rank) covers values of a set with a defined order relationship.
- *Numerical data* (quantitative) encompasses values of an ordered set with predefined operations (an algebraic field). It can be further distinguished according to the density of values in the discrete ($S \in Z$) or continuous ($S \in R$) set.
- *Interval data* refers to intervals instead of single values.

	Nominal	Ordinal	Numerical	Interval
Delta Operators [BSBF02]			√√	√
Buffer [dRRM06]	√√	√√	√√	√√
Warm-up Time [BHRE07]	√√	√√	√√	√
Context Regions [APJ+03]		√√	√√	√
Sensitivity (speed, acceleration) [PZLB05]			√√	√
Switch [PZLB05]	√√	√√	√√	√√
Statistical Techniques (average, variance)			√√	√
Filtering Techniques				
Statistical Filtering			√√	√
Parametric Filtering (time, localisation)	√	√	√√	√
Threshold techniques				
Simple threshold		√√	√√	√
Double threshold		√√	√√	√
Double double threshold (hysteresis) [TC04]		√√	√√	√
Refresh techniques				
Parametric Refresh (T)	√√	√√	√√	√√
Event/Action Refresh	√√	√√	√√	√√
Probabilistic schemas				
Fuzzy Logic [Dar07]	√√	√√	√√	√√
Markov Chain [SAH07]		√√	√√	√
Bayesian Network [SAH07]			√√	√
Dempster-Shaffer Theory (DST) [SAH07]			√√	√

Table 5.1: Characterization of Stabilization Algorithms According to the Data Scope Criterion.

Table 5.1 reports an overall characterization of stabilization mechanisms according to the *data scope*. We use the annotation "√√" to express that an algorithm completely targets the *data scope*, while "√" expresses that an algorithm only partially targets *data scope*.

The *data scope* gives an hint to developers about the scope of information that stabilization algorithms expect as an input value. They can therefore knowledgeably choose the one that meet their needs.

Algorithmic Responsiveness

Algorithmic responsiveness characterizes the responsiveness of a stabilization algorithm. The responsiveness of a stabilization algorithm, is the time that elapses between the reception of an input signal (information) by an algorithm, and the output signal that the latter delivers to the system in response. The system here can be any software application that implements stabilization mechanisms. In particular, in the context of this chapter it refers to the control

Algorithmic Process Behavior Class	Analytic Definition
$Class\ T_1$	$T_1(n_t) = K$, with K constant
$Class\ T_2$	$T_2(n_t) = f(n_t)$
$Class\ T_3$	$T_3(n_t) = g(n_t) + \sum_{i \in [0,t^*[} g(n_i)$

Table 5.2: Analytic Definition of Algorithmic Process Behavior Classes

system.

The algorithmic responsiveness is correlated to *algorithmic execution time* which impacts the *performance* of the overall system. When *algorithmic responsiveness* is *high*, *algorithmic execution time* decreases and the overall reactivity of the system is improved. *Algorithmic responsiveness* can shepherd developers of control systems in order to asses the impact of the choice of an algorithm on the performance of the system.

The responsiveness of an algorithm evaluates the latency of producing an output signal from a given input. To be able to reason on *algorithmic responsiveness*, we must define a property that enables us to asses it for a given algorithm. For that purpose, we introduce the notion of *algorithmic process behavior* which characterizes the processing time for a given algorithm. The processing time corresponds to the time of execution required for an algorithm to produce an output signal for a given input signal.

We defined three classes of algorithmic process behavior. Table 5.2 summarizes the analytic description of these classes. The first class T_1 corresponds to algorithms with a constant process time that does not depends on the size of input data. The second class T_2 corresponds to algorithms which process time is a function of data input size. The third class T_3 describes algorithms for which the process time is a function of data size and the time of the learning internal process.

Using the analytic definition of algorithmic classes defined in Table 5.2, stabilization algorithms can be intuitively mapped to the corresponding class of process behavior. Algorithmic process behavior classes allow assessing the algorithmic responsiveness. This means that, if the process classes of two algorithms are known, we can deduce which of them has the higher responsiveness. However, to be able to compare the responsiveness of stabilization algorithms, it is necessary to explicit the definition of the functions $f(n_t)$, $g(n_t)$ of the corresponding class. For example, assuming $f(n_t)$, $g(n_t)$ to be constant, we can deduce that $T_3(n_t) \leq T_2(n_t)\ for\ t^* \leq \frac{k_2 - k_3}{k_3^{'}}$, with $k_2, k_3, k_3^{'}$ constants. That means that, for constant behavioral functions the responsiveness of an algorithm of class T_3 is less than that of class T_2 when the execution time is better than $(\frac{k_2 - k_3}{k_3^{'}})$.

In practice, the analytic expression of functions $f(n_t)$, $g(n_t)$ can be derived empirically using analysis method like *least square regression*. However, it is not always easy to perform such empiric experiments because, an algorithm can have different expressions corresponding to various range of data size. In addition, it is pretty hard to define threshold values for data size range for which a given functional expression is valid.

For the foregoing reasons, we define an association between algorithmic process behavior classes and the following qualitative attributes: *Low*, *Medium* and *High*. We also introduce the notion of initial *cost*, C_{init}, which corresponds to the *initial responsiveness*. The *initial responsiveness* can be defined as the time elapsed between the time t_0 of the reception of the first signal, and the time t^* when the algorithm produces the first valid output.

From empirical observations of stabilization mechanisms listed in our classification, we noticed that the cost $C_{init}(T1) \leq C_{init}(T2) \leq C_{init}(T3)$. That is, the initial *cost* of algorithms of class T_1 is less or equal to the *cost* of algorithms of class T_2 that is less or equal to the *cost* of algorithms of class T_3.

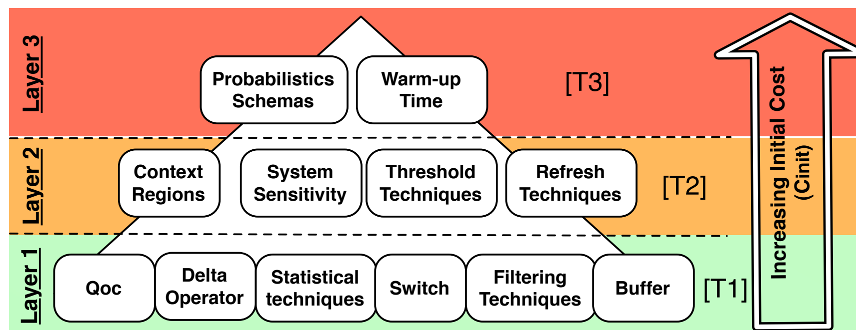

Figure 5.10: Classification of Stabilization Algorithms According to their Class and Cost

Figure 5.10 depicts the classification of some common stabilization mechanisms on the basis of their class characterization and the initial cost. The figure presents a multi-layered diagram, that describes the relationships between stabilization algorithms. The diagram consists of 3 layers. *Layer 1* is associated with class T_1 and *Low* initial *cost*, *Layer 2* is associated with class T_2 and *Medium* initial *cost* and *Layer 3* is associated with class T_3 and *High* initial *cost*.

Considering two algorithms A_i and A_j defined respectively with the cost C_i, C_j and class $T_i, T_j \, with \, i, j \in [1, 3]$; if $C_j < C_i$, then A_j has a higher initial responsiveness than A_i. However, it is important to notice that over the time, as the system is running, the responsiveness of A_i can increase significantly and become better than that of A_j.

Figures 5.11 *a-b-c*, give a detailed description of the algorithms that compose each layer of Figure 5.10. We choose to represent relationships between stabilization algorithms using features diagrams [KCH+90] notation. Feature diagrams allow the expression of similarities and choice options between stabilizations mechanisms. Figures 5.11 *a-b-c* represent stabilization mechanisms as feature elements. For example, on Figure 5.11 *a*, the diagram points out a dependency relationship between "*Switch*" and any other algorithm of the layer. Similarly, an exclusion relationship between "*Context Region*" and "*Double-double*" is depicted in Figure 5.11 *b*. Finally, an exclusion relationship is drawn between stabilization mechanisms that implement quite similar methods of data processing.

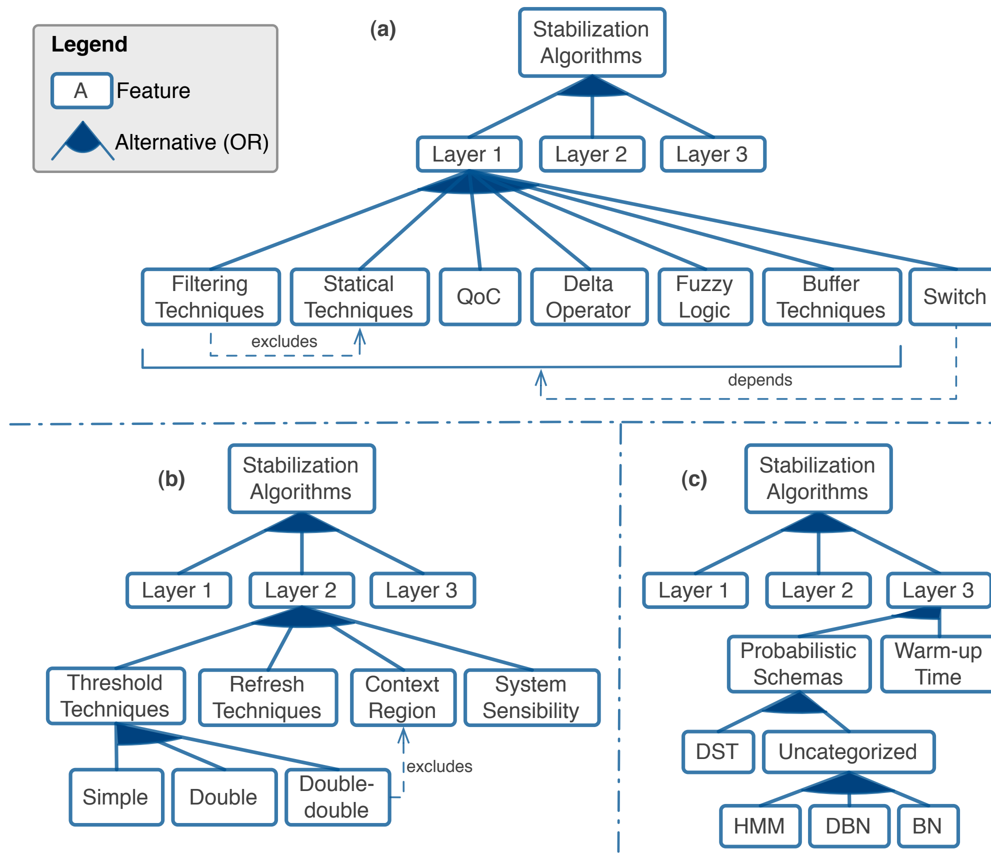

Figure 5.11: Relationships between algorithms of the first(a), second (b) and third (c) layer of the classification

Composition of Stabilization Algorithms

On an individual basis, stabilization algorithms offer interesting properties in order to stabilize data processing within the feedback control loop. For example, learning-based algorithms like *Kalmann Filter* are able to take into account the data history in order to predict the future, and threshold-based algorithms like *simple threshold* are able to detect small variations between consecutive data sets. When composing them adequately, stabilization algorithms offers more interesting properties that result from the combination of the properties of each algorithm participating in the composition. In particular, we are going to focus here on three types of composition model: The *horizontal*, the *vertical*, and the *hybrid composition* model.

Horizontal Composition

Although learning-based stabilization algorithms, like *Dempster-Shaffer Theory* (DST) [Wu03] or *Bayesian Networks* (BN) are efficient in predicting contextual changes, they introduce a latency in detecting variations in the application environment. According to our characteri-

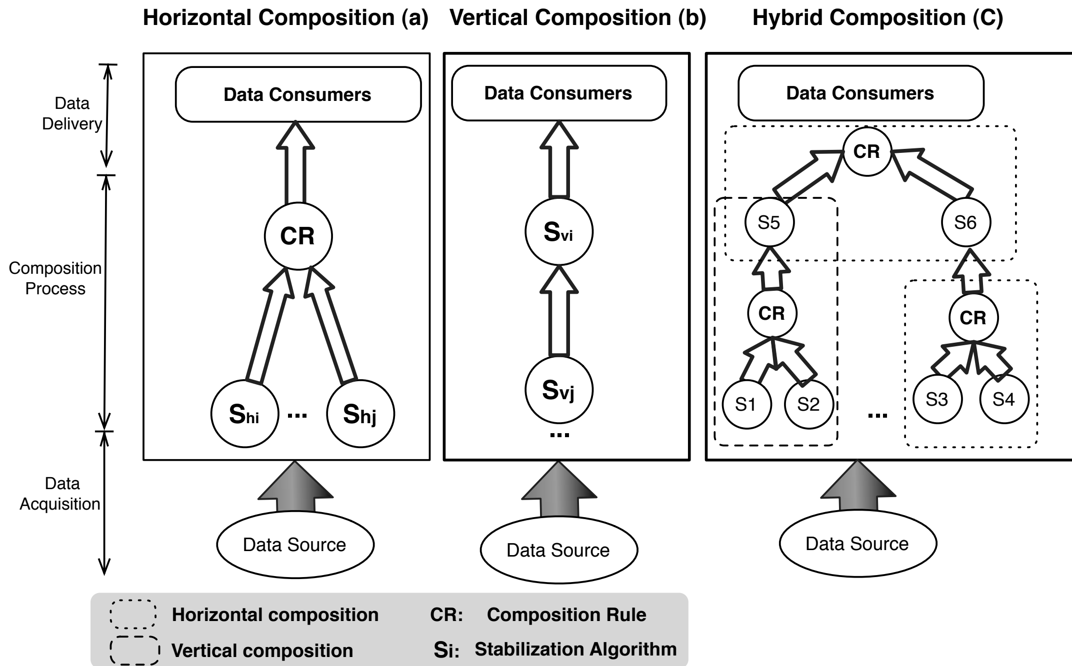

Figure 5.12: Stabilization Algorithms Composition Models

zation (c.f Section 5.3.2), these algorithms correspond to the class T_3 with a high initial cost. On the other hand, algorithms based on threshold evaluation functions [BSBF02] are more reactive than learning-based algorithms and are associated to the class T_1 with a low initial execution cost. We can achieve an efficient stabilization process by combining both classes of algorithms through appropriate composition rules. One way to do this, is to use of the principle of the horizontal composition, which consists in the *concurrent* execution of two or more stabilization algorithms of *different classes*.

For example, using horizontal composition the latency of learning-based algorithms can be compensated by the reactivity of threshold-based algorithms. The composition rule can be a simple rule like "$f(v_n, v_{n+1}) = max(v_n, v_{n+1})$", where v_n, v_{n+1} are context values, or a more complex rule involving *Quality of Context* (QoC) of the processed data. In practice, composition rules can be expressed with existing rule-based frameworks, like JESS [jes] or Drools [dro]. Figure 5.12-*a* gives an illustration of the horizontal composition strategy. On the figure, the output of *source data* is concurrently processed by two algorithms of different classes S_{h_i}, S_{h_j}. Then, outputs of these algorithms are selected or moderated according to the defined composition rule (CR) before being forwarded to *data consumers*.

Vertical Composition

The vertical composition consists in a sequential execution of stabilization algorithms organized as a stack. In a vertical composition, when composed algorithms have the same initial cost, they can be placed at any position of the stack. On contrary, when the initial cost is

different, algorithms with high initial cost will be located at the top of the stack. These association rules are justified by the fact that, the amount of processed data decreases from the bottom to the top of the stack, as they are being filtered out at each step. Thus, algorithms with a high initial cost at the top of the stack would have to process less data than algorithms at bottom. In addition, this organization reduces the overall cost of the stabilization process at least at the beginning of the stabilization process.

Vertical composition can be used to improve the accuracy of data processing, by a sequential refinement of the information through the stabilization stack. Figure 5.12 *b* illustrates the vertical composition strategy. On the figure, the output of the data source is sequentially processed through the stack of stabilization algorithms before being delivered to data consumers.

Hybrid Composition
The horizontal and vertical composition strategies can be combined using composition rules, to achieve complex and richer behavior of the stabilization process. *Hybrid composition* consist in the combination of horizontal and vertical composition models. Figure 5.12-*c* illustrates the combination of both strategies. This combination can be used in order to meet accuracy and efficiency properties of the stabilization process.

The data source in Figures 5.12 *a-b-c* stands for any data provider, since the stabilization strategies can be applied on any data coming from monitoring activities and on data coming from adaptation activities like the decision-making entities.

5.4 Summary

In this chapter we have presented our solution for supporting the reification of feedback control loop architecture elements as first-class citizens at runtime. Feedback loop architecture elements are implemented by SCA components, which enables their instrumentation for verification or validation purposes. We have also, showed how the feedback control loop can be customized with cross-cutting concerns in our approach. In particular, we focused on the implementation of a stable decision-making in the feedback control loop. We argued that stabilization mechanisms were an effective method for implementing stable feedback control systems. The specialization of the feedback control system through stabilization mechanisms is based on a composition model which enables to improve the accuracy or the latency of the control system.

The *CORONA* approach is cost-effective and reduces the burden of implementing autonomic systems for software engineers. That is because, *CORONA* supports the generation of the control system implementation code from a control architecture description. In addition, one of the originality of the *CORONA* approach revolves in the fact that, it tries to promote good practices of engineering control systems inherited from the field of control theory. In particular, the implementation of stabilization mechanisms in order to reach stable decision-making.

In the next chapter, we will introduce the compilation infrastructure of the *CORONA* toolchain. We will principally focus on two aspects: First, on the mapping rules that guide the generation of the control system implementation. Especially, rules or constraints related to the description of the interaction between control elements. The second aspect we will pay attention to, is the verification of the control system architecture. In particular, conflicts checking in order to ensure coherence of the control system in the context of multiple feedback control loops architectures.

Chapter 6

Compilation Infrastructure

"It is unworthy of excellent men to lose hours like slaves in the labor of calculation which could be relegated to anyone else if machines were used."

– Gottfried Leibnitz

Contents

6.1 Component-based Generative ToolChain . . . 82
6.1.1 CORONA Compiler . . . 83
6.1.2 Location Optimizer . . . 84
6.1.3 Verification Generator . . . 84
6.2 Mapping from SALTY Model to SCA Model . . . 85
6.2.1 Mapping Rules . . . 85
6.3 Control Loop Architecture Distribution . . . 89
6.4 Conflicts Checking on Feedback Control Loop Architectures . . . 93
6.4.1 Motivating Example . . . 94
6.4.2 Conflicts Pattern Modelisation . . . 95
6.4.3 Conflicts Verification Algorithms . . . 99
6.4.4 Conflicts Resolution . . . 102
6.5 Control Loop Architecture Evolution . . . 105
6.6 Summary . . . 106

In Chapter 5, we have presented how feedback control loops can be reified at runtime in order to increase their visibility. In particular, we have discussed the implementation of the feedback control architecture in SCA. We have drawn the relationship between control theory and software engineering, and introduced some properties of feedback control systems from control theory. We focused on the *stability* property, and suggest a methodology to address this property when implementing control systems in software engineering.

The current chapter puts the focus on the compilation infrastructure of the *CORONA* toolchain. This chapter investigates principally two challenges: The first challenge is the *mapping challenge*, which consists of filling the gap between the expressiveness of architectural description languages, and the implementation platform languages. The second challenge that is addressed in this chapter is the *verification challenge*, which aims at benefiting from the explicitness of the control system architecture for building verification tools that help developers to build autonomic systems that are more reliable. These two challenges are addressed by the *CORONA* toolchain. The first challenge is addressed by the *CORONA* toolchain through the generation of the runtime implementation code of the control architecture from an architectural description. That is the toolchain automatically maps design concepts into runtime concepts. The second challenge is addressed by the *CORONA* toolchain through the implementation of a verification algorithm that checks the control architecture against some conflicts between feedback control loops in the control system.

Structure of the Chapter

The rest of this chapter is organized as follows: we start by introducing features of the *CORONA* toolchain that make cost-effective the implementation of autonomic systems (cf. Section 6.1). Then, we elaborate in details on each of these features. In Section 6.2 we discuss the mapping relationship between the SALTY design model and the SCA runtime concepts. The deployment issue of the feedback control loop is discussed in Section 6.3. In Section 6.4, we present some verification algorithms for the detection of conflicts in the control system architecture. Section 6.5 addresses evolution aspects of the control system. Finally, we conclude this chapter with a summary (cf. Section 6.6).

6.1 Component-based Generative ToolChain

In Chapter 4– Section 4.3.2, we have presented an overview of the toolchain *CORONA*. In this section, we are going more into details and discuss main features of the toolchain. Figure 6.1 gives an insight of these main features.

Figure 6.1 depicts the three main features of the toolchain *CORONA*: The *compiler*, the *location optimizer* and the *verification generator*. The *location optimizer* and the *verification generator* play the role of tool support, and provide design feedbacks to architects/developers of autonomic systems. When the control system architecture changes or evolves, the *compiler* takes into account these changes and generates the corresponding source code.

The toolchain and its components are implemented according to the SCA standard. This enhances the flexibility of the toolchain, and provides means to extend it with few engineering efforts. Now, let us take a close look at the architecture of key features of the *CORONA* toolchain.

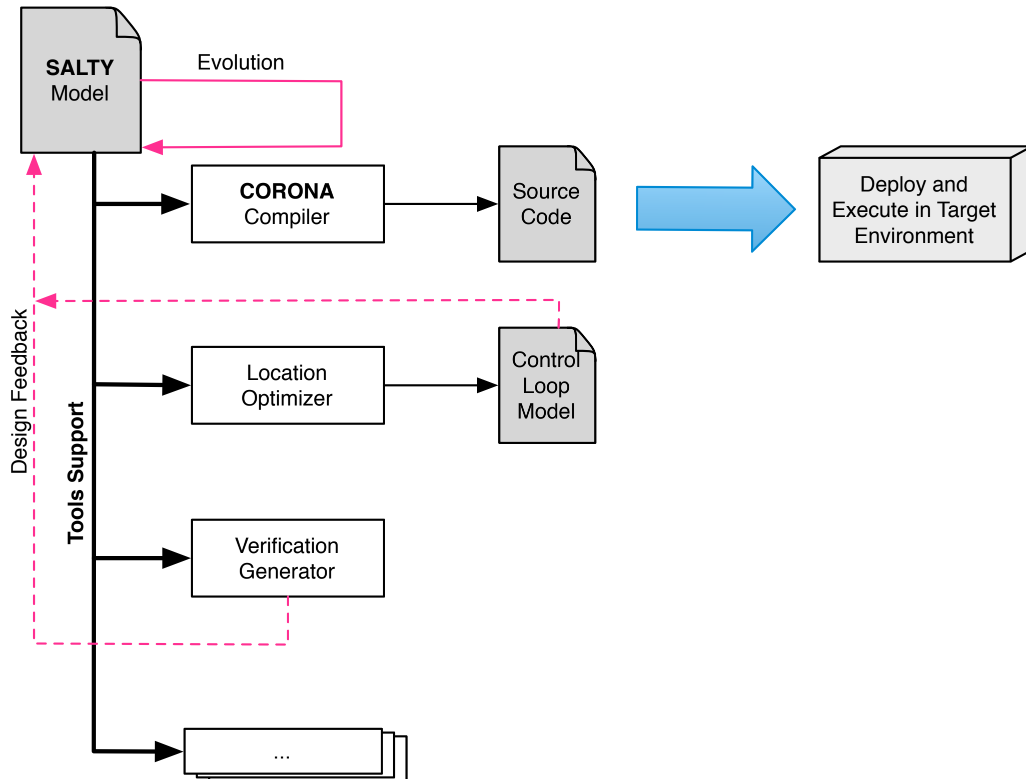

Figure 6.1: ToolChain Key Features Behavior

6.1.1 CORONA Compiler

The *compiler* in the *CORONA* toolchain is responsible for two main activities. First, the mapping between the control loop architecture model and the target platform model. Second, the *evolution* of the control architecture to ensure the consistensy between the generated source code and the control system architecture.

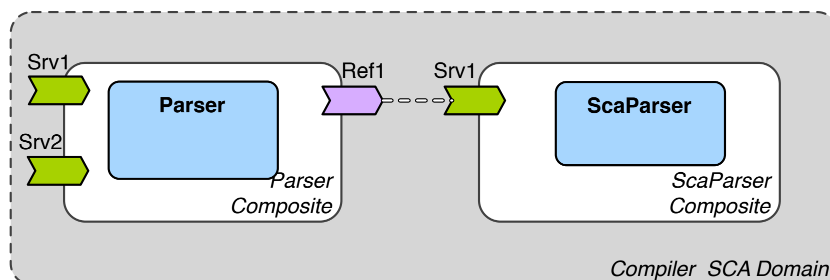

Figure 6.2: SCA Architecture of the *CORONA* Compiler

Figure 6.2 gives the SCA architecture of the compiler. The compiler consists of two composites: the *Parser* and *ScaParser* composite. The composite *Parser* has two services (*Srv1*,*Svr2*), and one reference *(Ref1)*. One of the service (*Srv1*) enables the loading process of the control system architecture model, and the second launches the generation process. The composite *Parser* has a dependency to the *ScaParser* composite for generating SCA implementation. To carry out another platform than SCA, it will be enough to extends the *Parser* composite with a dependency reference for the new target platform. The service-oriented architecture of the compiler enforces the flexibility of the *CORONA* toolchain, by providing extension capabilities such as support for multiple target platforms.

6.1.2 Location Optimizer

The *location optimizer* provides a tool to developers of autonomic systems to tackle the issue of the distributed deployment of the control loop. The *location optimizer* uses Constraint Satisfaction Problem (CSP) techniques [Apt03b] to assign control elements among available resources of the managed system. The assignment of a control element to a specific host resource, is done by enriching the architectural model of the control loop with adequate annotations. The *location optimizer* tool can be used for example, in the context when the developer wants to optimize the distribution of the control loop components at runtime, knowing the network topology of the deployment infrastructure. The optimization of the distribution through the *location optimizer* can be done according to criteria like the bandwidth between host machines.

The *location optimizer* tool is implemented as an SCA composite. This tool is not automatically trigger for a given control loop architecture, but must be explicitly called. It requires the control loop architecture model and the network topology model as input, and generates a control loop architecture model enriched with annotations as output. This architecture model contains annotations that drive *CORONA* compiler during the generation of the source code, and the deployment script. We elaborates in details on the *location optimizer* through a concrete example on Section 6.3 below.

6.1.3 Verification Generator

The *verification generator* is a tool that implements a set of algorithms that analyze the control architecture. In particular, algorithms for detecting conflicts in the control architecture. The *verification generator*, provides feedbacks to developers when some conflicts are found in the control system architecture.

The verification generator is implemented as an SCA composite. It takes as input the architecture model of the feedback control loop. This tool is automatically triggered by the *CORONA* toolchain before the generation of the source code. However, even if it is strongly recommended, developers can decide to follow or not the warnings generated by this tool without prejudices for the generation of the source code. We elaborate in details on the *Verification generator* on Section 6.4.

6.2 Mapping from SALTY Model to SCA Model

One of the challenges that need to be addressed concerning the engineering of autonomic system, is to fill the gap of expressiveness between control architectural description languages and target implementation languages. Indeed, architectural languages empower developers with domain specific concepts that make their life easier, because these concepts are close to their application domain. Inversely, generic implementation languages provide developers with a bit too generic concepts that are far away from their application domains. In our approach, we use MDE techniques to map the SALTY architectural language to the SCA implementation language. In this manner, developers of control systems can benefit from the expressiveness of the SALTY model without worrying about the mapping into a runtime implementation language.

The mapping between SALTY model and SCA language is challenging, because the concepts manipulated in both languages are different. For instance, the *ecore* model of SCA core package has 55 classes, while SALTY model consists of 90 classes. In this section, we are not going to provide a mapping between these two models. In particular, we are going to elaborate on four different aspects of the mapping, and namely: the composition, the dynamic, the implementation and interaction aspects.

6.2.1 Mapping Rules

Before we dive into the details of mapping rules, we first introduce an example of a control architecture specified with the SALTY model. Figure 6.3 gives an example of the Condor control system architecture. The Condor control system architecture is composed of two feedback control loops: The *Main* feedback control loop and the *SubmissionRate* feedback control loop. Both loops have an hierarchical dependency. The *SubmissionRate* FCL is controlled by the *Main* FCL. The Condor control system is used to manage jobs arrival rate into the queue of the Condor infrastructure [TTL05].

In this section, we are not focusing on the explanation of the workflow of the Condor control system architecture. This workflow is discussed in details in the validation Section (c.f Part III) of this thesis. Here, we use the Condor architecture to illustrate the main concepts of the SALTY model. In particular, in order to exemplify the mapping rules between SALTY and SCA concepts. Now, let's discuss the mapping between the SALTY and the SCA model.

A) Composition Aspect

The expression of the composition in the SALTY language can mainly be done through two constructions: *provided* feature, and *composite*.

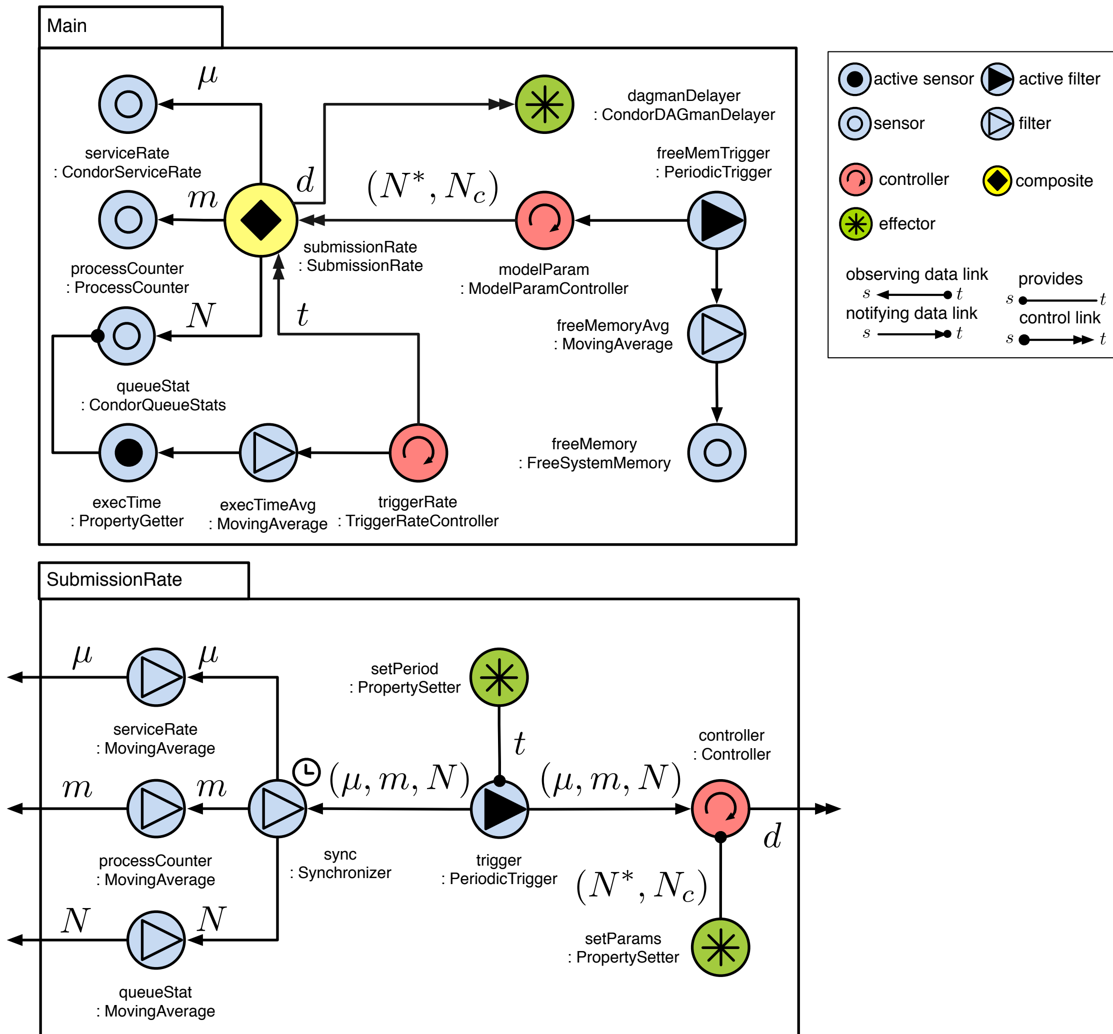

Figure 6.3: Condor [TTL05] Control Loop Architecture

Provided features enable the construction of compound control elements. A provided feature can be a sensor or an effector. On Figure 6.3, the sensor *queueStat* is a compound control element with one provided element, the sensor *execTime*. Similarly, the filter *trigger* has a provided effector, *setPeriod*. Compound control elements with provided features are mapped into SCA composites, where each provided element is represented as a SCA component.

Composites in the SALTY language are used to create individual control loops or to build compound control elements. On Figure 6.3 we have two composites: *SubmissionRate* composite and *Main* composite. These composites are mapped into SCA composites.

B) Dynamic Aspect

Dynamic aspect refers to the dynamicity of the information flow between the nodes of the control system.

Active control elements initiate the propagation of information between them and connected control elements. Information is pushed with a frequency rate. we distinguish *active* filters and sensors in the SALTY language. Active control elements are mapped into SCA components or composites with a property that represents the frequency rate at which information is pushed. On Figure 6.3, the filter *trigger* and the sensor *execTime* are example of active control elements.

Passive control elements are passive to the propagation of the information. This means that, connected control elements need to initiate the process of collecting information they require. On Figure 6.3, *serviceRate* and *freeMemoryAvg* are examples of passive control elements. Passive control elements are mapped into SCA components or composites depending on whether or not they are compound control elements.

C) Implementation Aspect

Control element types are mapped to a Java class that implements the behavior of the corresponding component. The component is related to other component of the control architecture through SCA services and references. On Figure 6.3, *TriggerRateController* indicates the type of the controller *triggerRate*.

Link types are mapped into SCA service or reference. The SALTY language distinguishes between two types of link: *control link* and *data link*. On Figure 6.3, *data links* are represented by simple arrow lines, and *control links* by double arrow lines.

D) Interaction Aspect

Interactions between control elements describe the information propagation flow in the control system. On Figure 6.3, interactions are represented by oriented arrow lines that connect control elements. Interactions are mapped into SCA binding or wires.

So far, we have seen that architects can use *data links* or *control links* to specify interactions between control elements. In Section 5.2.5, we have described the three modes of interaction between the nodes of the control system architecture in SALTY: The notification, the observation and the multimodal mode. However, the specification of interactions can sometimes leads to ambiguous execution semantic.

A good illustration of the ambiguity that can occur is illustrated by Figure 6.4. The figure depicts *simpleFilter* with three connections links: two *Push* link and one *Pull* link. This architectural representation leads to an ambiguity concerning the semantic of the execution

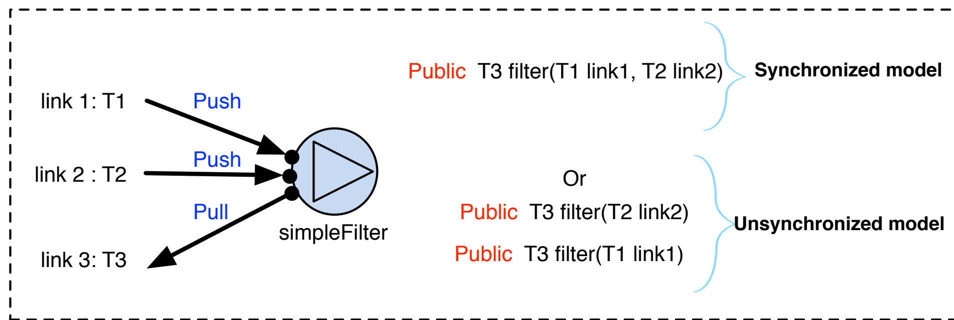

Figure 6.4: Illustration of Possible ambiguity in the interaction model

SALTY Concepts	SCA Concepts
Simple Control Elements	Component
Compound Control Element	Composite
Provided Features	Component
Composite	Composite
Control System	Composite(s)
Active Control Element	Component/Composite with a property
Control Element Type	Java Class
Control Link	Service/Reference
Data Link	Service/Reference
Bind	Wire/Binding
DataType	Java Class or primitive Java type

Table 6.1: Mapping Rules Between SALTY and SCA Concepts

of the *simpleFilter*. This is expressed by three different signatures of the method *filter()*, that implements the business logic of the *simpleFilter*. The semantics of the first signature suppose that the *simpleFilter* should wait for the two values from *link1* and *link2* before computing the value that is pulled out on *link3*. This signature corresponds to the synchronized interaction model. The second and the third signature suppose that the filter computes the output value (*link3*), upon receiving any value from *link1* or *link2*. This means that only one of the link is required for the calculation of the output value. The last two signatures correspond to the an unsynchronized interaction model.

To leverage the ambiguity of the semantics of interactions between control nodes, we have adopted the *synchronized* interaction model. In this model, each control element requires all input values for calculating the output. This means that, on the example of Figure 6.4, the first method signature corresponds to the execution semantics. Table 6.1 summarizes the mapping rules between SALTY and SCA concepts.

In this section, we have given an insight of the architecture description semantics, and

the corresponding runtime concepts in SCA. In the next Section 6.3, we discuss how the distributed deployment of the control architecture is tackled in our approach.

6.3 Control Loop Architecture Distribution

As previously introduced, thanks to its component-based architecture, *CORONA* provides the flexibility to integrate customized services in order to check or refine control architectural models. In particular, *CORONA* integrates *location optimizer* for calculating the distribution of control elements on the deployment infrastructure, with respect to some constraints objectives. One of the constraint can consist for example in ensuring a minimal communication latency between distributed control elements. In this the case, the *location optimizer* strives to assign the deployment of control elements on hosts with high bandwidth.

The *location optimizer* service uses the control system architecture model, the network model of the deployment infrastructure, and constraints objectives to determine the best distribution of control elements. The control system architecture model is defined using SALTY specifications. The network model is defined from the network meta-model, and constraint objectives are specified using a CSP API. The current implementation of *location optimizer* is based on the *JaCoP* (*Java Constraint Programming*) solver [KS10] API. In the following sections, we discuss how the network model and constraints are specified for the *location optimizer* in *CORONA*.

Network Model

The network model includes information about host machines, the relationships between them, as well as additional properties such as supported protocols, available memory on machines, and the storage capacity. Figure 6.5 gives an overview of the main concepts of the network meta-model. The figure shows that the *network* infrastructure consists of a set of *hosts* with properties and communication *protocols*. For each protocol supported by a host, we can specify *users* credentials for accessing that host. The network model is specified independently from the control loop architecture model, and provides information about the deployment infrastructure.

Once the control architecture model and the network model are provided, the *location optimizer* is almost ready to run. However, to have it completely ready to run, developers must specify constraints to configure its behavior. These constraints are defined through the objective function and constraint annotations.

Objective function

Prior to the definition of the objective function of the *location optimizer* service, let us define a set of concepts that are used for implementing its logic behavior.

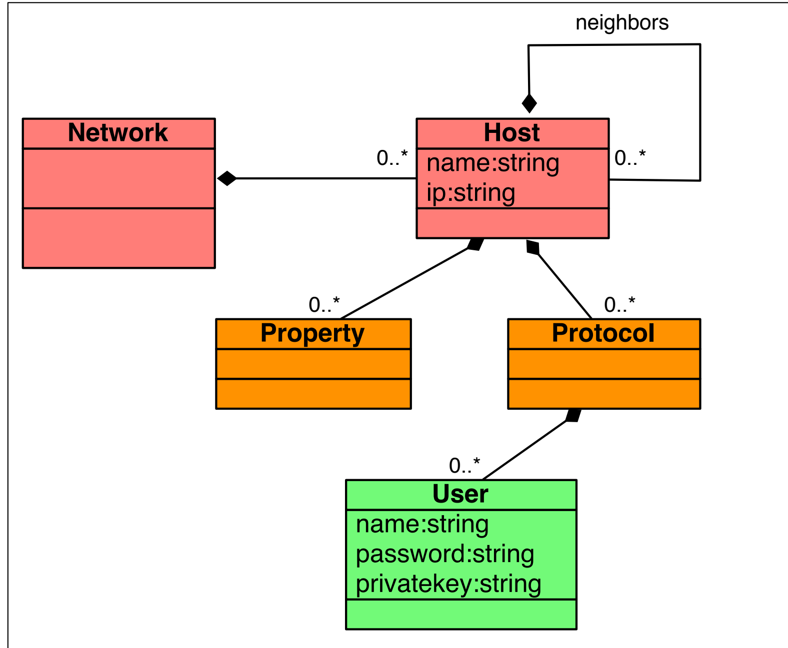

Figure 6.5: Basic Concepts of the Network Meta-Model

Core Definitions

For sake of simplicity, control elements–(*sensor, effector, filter, controller*)– in these definitions are subsumed under the term of *nodes*. The control system (CS) represents the control system architecture. These definitions characterize the control architecture and the deployment infrastructure network.

Def. 1. $\mathcal{CS} = \{l_1, \ldots, l_i, \ldots, l_n\},\ 1 \leq i \leq n \wedge 1 \leq n$: *denotes the different loops that compose the control system.*

Def. 2. $\mathcal{E} = \{e \mid \exists N_1, N_2 \in \mathcal{N}\ e = \{N_1, N_2\}\}$: *represents the connections between nodes of the control system.*

Def. 3. $\mathcal{N} = \{N_1, \ldots, N_i, \ldots, N_m\}, 1 \leq i \leq m$: *The nodes of the control loop architecture.*

Def. 4. $\mathcal{H} = \{H_1, \ldots, H_i, \ldots, H_n\}, 1 \leq i \leq n$: *The host machines of the deployment infrastructure.*

Def. 5. $\mathcal{CH} = \{(H_i, H_j) \in H^2 : H_i\ connected\ H_j\}, i \neq j \wedge 1 \leq i \leq n \wedge 1 \leq j \leq n$: *Connections between the hosts of the network.*

Def. 6. $\mathcal{NDH}_j = \{N_1, \ldots, N_i, \ldots, N_o\}, 1 \leq i \leq o \wedge o \leq m$: *Nodes deployed on the jth host.* This set is a subset of $\mathcal{N}$.

Def. 7. $\mathcal{DH} = \{H_1, \ldots, H_i, \ldots, H_p\}, 1 \leq i \leq p \wedge p \leq n$: *Hosts selected for the node deployment.* The set is a subset of $\mathcal{H}$.

Additionally, we define the following functions that help us to indicate selected hosts for the deployment, and the assignment of a node to a specific host:

Def. 8. $(sh(H_i) = 1 \Rightarrow H_i \in \mathcal{DH}) \wedge (sh(H_i) = 0 \Rightarrow H_i \notin \mathcal{DH})$: *Indicates if the ith host is selected or excluded for the deployment.*

Def. 9. $(nidh_j(N_i) \Rightarrow N_i \in \mathcal{NDH}_j) \wedge (\neg nidh_j(N_i) \Rightarrow N_i \notin \mathcal{NDH}_j)$: *Indicates if the ith node is deployed on the j^{th}host.*

The objective function of the *location optimizer* is to find the minimal set of hosts which complies to the constraint objectives. Using the definitions presented above, the objective function of the *location optimizer* can be formalized as follow:

$$\text{MIN}\left(\sum_{i=1}^{i=|\mathcal{H}|} sh(H_i)\right) \quad (6.0)$$

Constraint Objectives

Constraint objectives are used to configure the behavior of the *location optimizer* service. These constraints can be specified at two levels: architectural and implementation level.

Implementation Level
Developers can specify constraints in the configuration Java class of the *location optimizer* service. This is done by defining primitives or set of constraints through the CSP application programming interface. For example, the following restrictions can be specified concerning the distribution of the nodes on the deployment infrastructure:

C1 $\{\forall\ N_i \in \mathcal{N} :\ (\exists H_j \in \mathcal{H} : nidh_j(N_i) = true)$: This restriction specifies that every node must be deployed on a host.

C2 $\{\forall\ N_i \in \mathcal{N} :\ (\forall H_j, H_k \in \mathcal{H} : nidh_j(N_i) = true \wedge nidh_k(N_i) = true \Rightarrow i = j)$: This restriction specifies that each host can host only one node.

C3 $\{\forall\ N_i, N_j \in \mathcal{N} :\ (N_i, N_j) \in \mathcal{E} \Rightarrow (\exists H_k, H_l, \in \mathcal{H} : (nidh_k(N_i) = true \wedge nidh_l(N_j) = true) \wedge (k = l \vee (k \neq l \wedge (H_k, H_j) \in \mathcal{CH})))$: This constraint specifies that two connected nodes are deployed on the same host or on two connected hosts.

Architectural Level
Developers can specify constraints on the control system architecture through annotations. Annotations can be specified on control elements or on their connections. In practice, two types of annotations can be used by developers. The *@Host(hostname)* annotation, and the *@Constraint(constraint logic)*. The *@Host* annotation is used to assign a node to a specific host

machine. It takes as parameter the name of the target host according to the network model specification. The *@Constraint* annotation is used to indicate constraints on a node, between two nodes, or on their connections. It takes as parameter the logic conditional constrained supported by the CSP specification. For example, the logical condition $N_i = N_j$ will be interpreted by the *location optimizer* service, as a constraint requiring the nodes N_i and N_j to be hosted on the same machine. The inverse of that expression–ie, $(N_i \neq N_j)$– will actually have the opposite meaning.

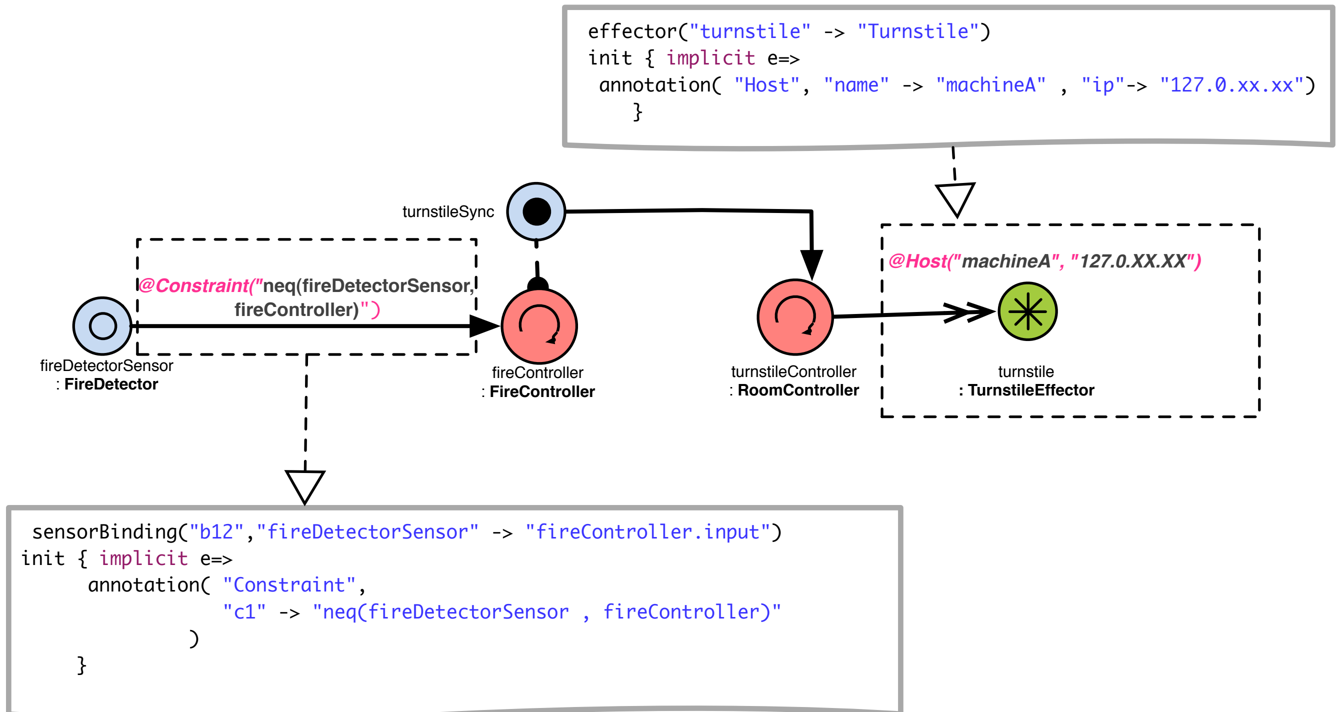

Figure 6.6: Illustration of a Feedback Control Loop Architecture with Constraints Annotations

Figure 6.6 gives an illustration of the specification of constraints on the control system architecture with the SALTY language. In fact, the language for expressing constraints is not provided by the initial SALTY language. But, thanks to the flexibility of the SALTY language, we can use the annotation construction to embed a CSP constraint syntax. The figure represents a control system architecture with one sensor *(fireDetectorSensor)*, two controllers *(fireController, turnstileController)*, and one effector *turnstile*. The distribution constraints are indicated for the *turnstile* effector, and the connection link between *fireDetectorSensor* and *fireController*. The annotation *@Host* on the effector indicates that it must be deployed on the host *machineA*. The annotation *@Constraint* on the connection link indicates that the sensor*(fireDetectorSensor)*, and the controller *(fireController)* must be deployed on different hosts. The figure also provides the corresponding syntax for specifying these constraints at the architectural level. It is important to notice that at the architectural level, constraints are in-

dicated for an *instance* of the control element, and not for its *type*. This means that we can have different *instances* of the same *type* of a control element with different constraints.

The execution result of the *location optimizer* is an architecture model where each node is annotated with information about the deployment. This information is indicated with the annotation *@Host*. For example, for a deployment environment that consists of two machines *A* and *B*, the execution of the *location optimizer* service on the control system architecture model presented on Figure 6.6 will produce the architecture model depicted on Figure 6.7.

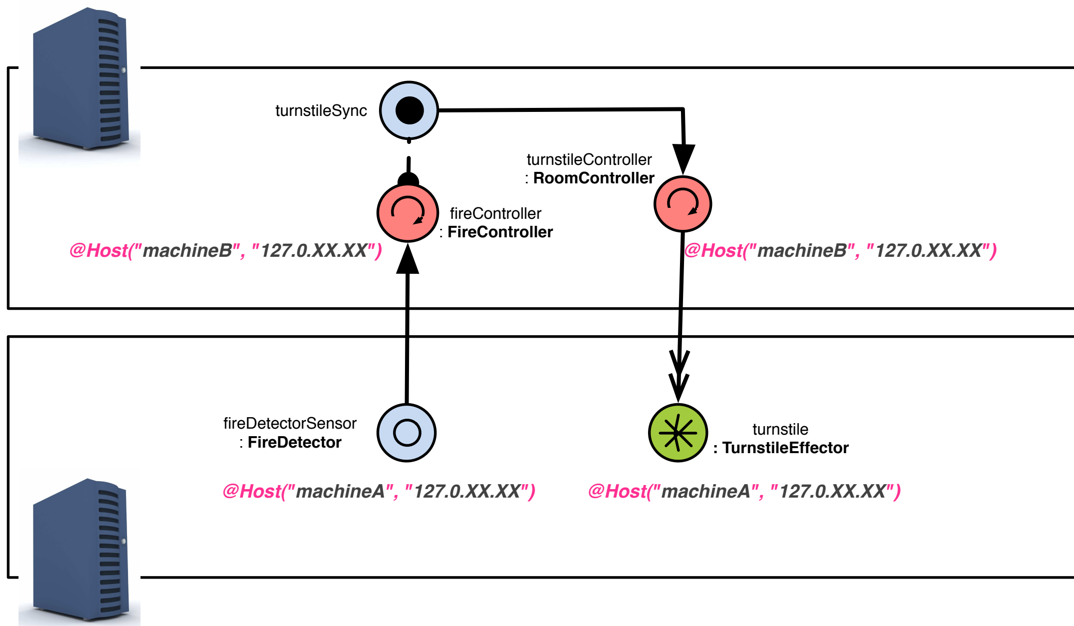

Figure 6.7: Computation Result of the location Optimizer Service

Figure 6.7 shows the result of the computation of the *location optimizer* service on a control architecture. The figure shows that the nodes of the feedback control loop are decorated with information about the deployment. This information drives the compiler for the generation of the source code and the deployment script.

In this section, we have explained how the distribution of the control system architecture was computed by the *location optimizer* service. We have also presented two ways of expressing constraint objectives for that service. In the next section (cf. Section 6.4), we discuss how conflict verifications are performed on the control system architecture.

6.4 Conflicts Checking on Feedback Control Loop Architectures

In this section, we discuss how the verification generator tool handles conflicts verification in the control system architecture. This section is organized as follows: In Section 6.4.1,

we introduce a motivation example to justify the importance of conflicts verification for the control system architecture. Then, in Section 6.4.2, we introduce the modeling of some conflicts patterns. Section 6.4.3 describes conflicts verifications algorithms implemented in the *CORONA* toolchain. Finally, Section 6.4.4 discusses the resolution of conflicts found in the control architecture.

6.4.1 Motivating Example

The visibility of feedback control loops at runtime enables the implementation of automated analysis tools that help reducing the cost of engineering autonomic systems. In particular, one important issue that software engineers face, is the unwanted or unaware interferences between control loops when designing complex control with multiple objectives. These interferences impact the behavior of the control system as a whole, and require a lot of efforts and time for their detection and correction.

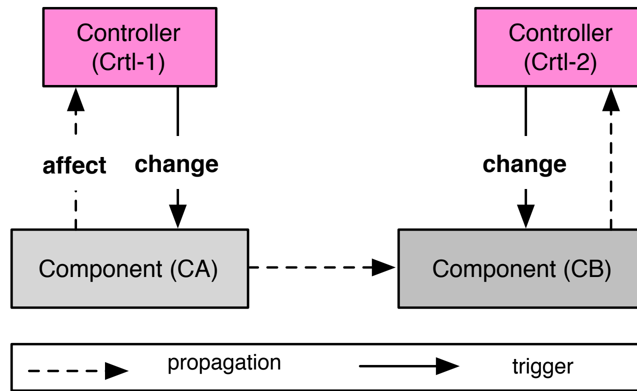

Figure 6.8: Illustration of the Invisible Interference Problem

A good illustration of that, is the *invisible interference problem* [HGB10]. The *invisible interference problem* is an indirect influence of one feedback control on another one. A controller triggers changes on a component, which triggers changes on a second component controlled by another controller. Figure 6.8 gives an illustration of the invisible interference problem.

Figure 6.8 depicts two controllers *Ctrl-1* and *Ctrl-2* interacting indirectly. When the controller (*Ctrl-1*) triggers changes on the component *CA*, the later propagates some changes on the component *CB*, which is observed by the controller *Ctrl-2*. Consequently, an action triggered by the controller *Ctrl-1* is propagated to the controller *Ctrl-2*. If the both controllers implement opposite objectives, then the system can turns into an unstable state.

Invisible interferences is a threat for the stability of the control system. In traditional engineering approaches, the implementation of the feedback control loop behavior is usually masked under layers of abstraction intended to hide the system complexity. The resulting lack of visibility makes difficult the analysis of the control system architecture and therefore

the detection of looming interference of the control flow. In the *CORONA* approach, the feedback control loop architecture is reify as first-class citizen which makes easier the analysis of control flows and the detection of potential interferences. Additionally, *CORONA* provides a strong mapping between the architectural model and target implementation models of the control system. This ensures the compliance of the architectural model and the generated implementation code of the control loop.

In general, the implementation of a control system begets many conflicts that are crucial to detect earlier, in order to guarantee the consistency of the control system. In the following sections, we introduce the types of conflicts (cf. Section 6.4.2) that are detected by the verification generator, then the algorithms of detection and correction (cf. Section 6.4.3) of some of these conflicts.

6.4.2 Conflicts Pattern Modelisation

The representation of feedback control loops as first-class entities at design and at runtime in the *CORONA* approach enables reasoning on control systems concerns. In *CORONA*, architecture analysis for conflicts detection takes place once architects have designed the FCL architecture. The result of this analysis is used to generate warnings when conflicts are found. At this stage, the architect can decide to modify the control system architecture in order to provide a new one that complies analyses, or to cancel generated warnings. By default, if warnings are not cancelled by the architect, the later are instrumented by the *compiler* for generating *supervisor* components for the runtime implementation of the control system. The role of *supervisors* is to coordinate the behavior of conflicting feedback control loops. *Supervisors* provide a correction for conflict patterns detected in the control loop architecture.

Figure 6.9 illustrates the verification of conflicts in the control loop architecture. This verification process consists of 4 main steps: *1) Architecture design*– on this step the control loop architecture model is provided as input to the verification generator tool. *2) Warnings generation*– algorithms for the detection of conflicts are run on the control loop model, and generate warnings when conflicts are found. *3)Warnings resolution*– the architect acknowledges the conflicts in the architecture and *resolve, cancel* or *ignore* them. *4)- Conflicts correction*– if some conflicts are ignored by the architect, the tool tries to resolve them by generating *supervisors*. The final architecture obtained at the end of the fourth step, is used for generating the implementation code of the control system.

Now that we have explained how the verification process is performed in *CORONA*, let us take a look at what type of conflicts can be detected through analysis of the control system architecture.

Conflicts Patterns

Conflicts in the control system architecture appears when feedback control loops overlap one another. More precisely, conflicts result from uncoordinated behavior of two or more control flows. In the example of *invisible interference* depicted on Figure 6.8, conflicts came from the

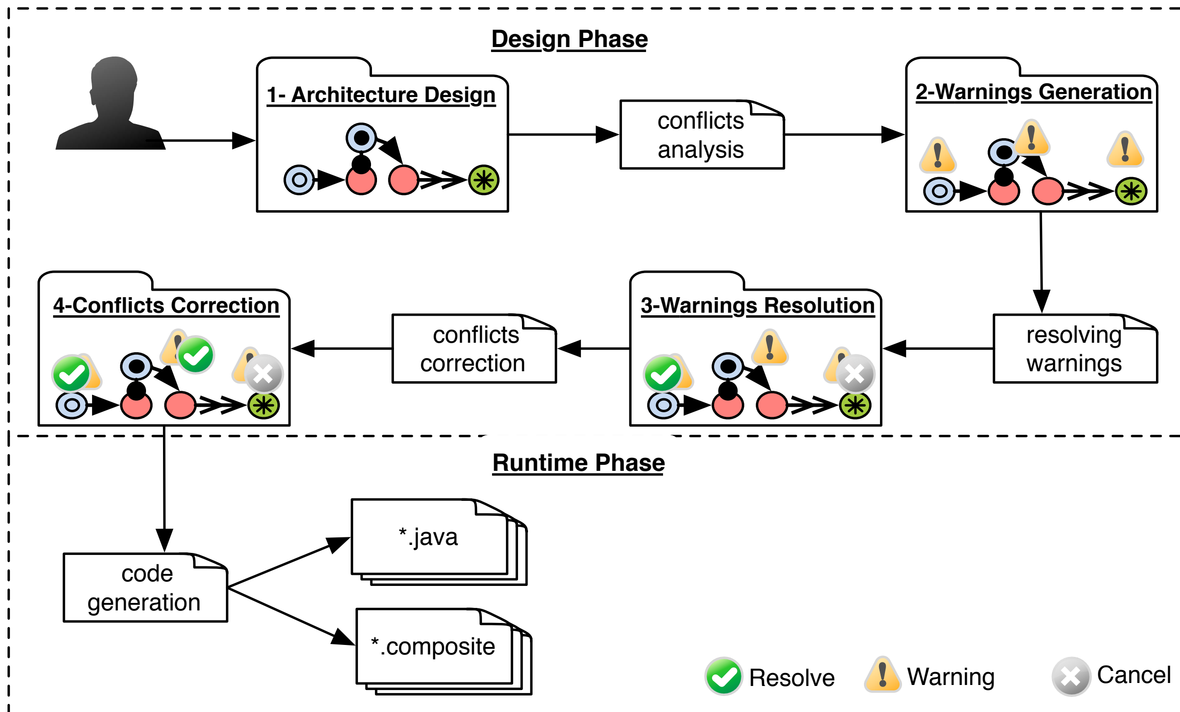

Figure 6.9: Conflicts verification Process

uncoordinated behavior of the control flow governed by the controller *Ctrl-1*, and the control flow governed by *Ctrl-2*. Indeed, in order to identify conflicts in the control architecture, we must be able to characterize these overlaps. We have identified two types of overlap in the control architecture: *Direct* and *Indirect* overlaps.

Direct overlap is an overlap between control loops that share at least one control element—i.e, $D_{ov}\ :\ \exists\, N_a, N_b\ \in\ l_1\,, N_c\ \in\ l_2\ |\ \exists\, e_1\ =\ (N_a, N_b)\quad e_2\ =\ (N_a, N_c)$. We distinguish four types of *direct overlaps*: *Sensor*, *filter*, *controller*, and *effector overlaps*.

Name: *Sensor Overlap*

Description: *Sensor overlap* characterizes feedback control loops that overlap at a *sensor* control element. Figure 6.10 gives an illustration of a sensor overlap between two feedback control loops *FCL4* et *FCL5*. These control loops have in common the sensor control element of type *SensorD*.

Problem: Potential conflicts of the control system can result from the fact that two feedback loops use the same source of information (*sensor*) to fulfill different actions. On the illustration of Figure 6.10, these actions are materialized by the effector *e5* of type *EffectorE*, and the effector *e4* of type *EffectorD*.

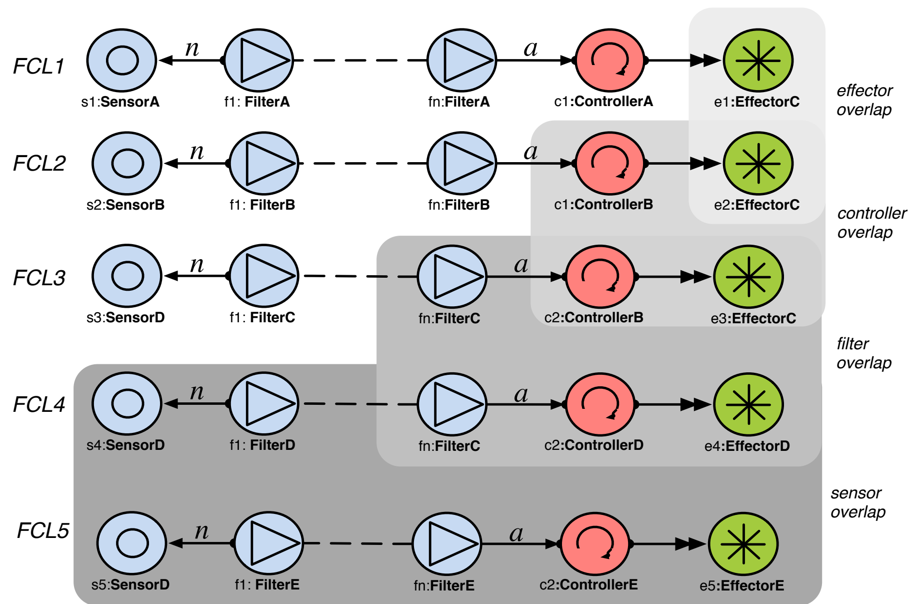

Figure 6.10: Direct overlaps Patterns

Name: *Filter Overlap*

Description: *Filter overlap* characterizes an overlap of feedback control loops at a *filter* control element. Figure 6.10 gives an illustration of a *filter overlap* between *FCL3* and *FCL4* on filter of type *FilterC*.

Problem: *Filters* have direct or indirect dependencies to *sensors*. Therefore, *filter overlap* indicates an implicit *sensor overlap*. *Filter overlap* can be of interest when two control loops share the same instance of a *filter*. In this context, when the filter is related to a *sensor* for example, the dependency between the sensor and the filter for one of the control loop will not be explicit, and so the *sensor overlap*. The importance of this pattern is well illustrated through the *transitive overlap* example on Figure 6.11.

Name: *Controller Overlap*

Description: *controller overlap* characterizes an overlap of control loop systems at a *controller* control element. Figure 6.10 depicts a *control overlap* between *FCL3* and *FCL2* on controller of type *ControllerB*.

Problem: When two controllers of two feedback control loops can take the same actions upon a managed system without any coordination, conflicting situations can arise. On the illustration of Figure 6.10, such conflicts can happen if the controllers *c1* and *c2* trigger actions with opposite behavior on the effector. An example of action with opposite behavior is a *start/stop* action.

Name: *Effector Overlap*

Description: *effector overlap* characterizes an overlap of control loop systems at an *effector* control element. Figure 6.10 illustrates an *effector overlap* between *FCL1* et *FCL2* on effector of type *EffectorC*.

Problem: The potential conflict results from the fact that two or more feedback control loops can take the same actions on a managed system, without any coordination. On the illustration example on Figure 6.10, we have two control loops *FCL1* and *FCL2* that implement different control policies (*ControllerA*, *ControllerB*) with the same action *(EffectorC)* scope on the system.

Indirect overlap denotes feedback control loops that indirectly share a control element. An example of an *indirect overlap* is a *transitive overlap*—*i.e.*, when one FCL overlaps a second FCL, which also overlaps a third one in such a way that there is an implicit dependency between control elements of the first and the third loop. The dependency between the first and the last feedback control loop is qualified as a *transitive* overlap. In a formal way, a *transitive overlap* (Tr_{ov}) can be defined as follows:

$$\exists\, l_1, l_2, l_3 \,|\, D_{ov}(l_1, l_2) = true\,, D_{ov}(l_2, l_3) = true \;\Rightarrow Tr_{ov}(l_1, l_3) = true.$$

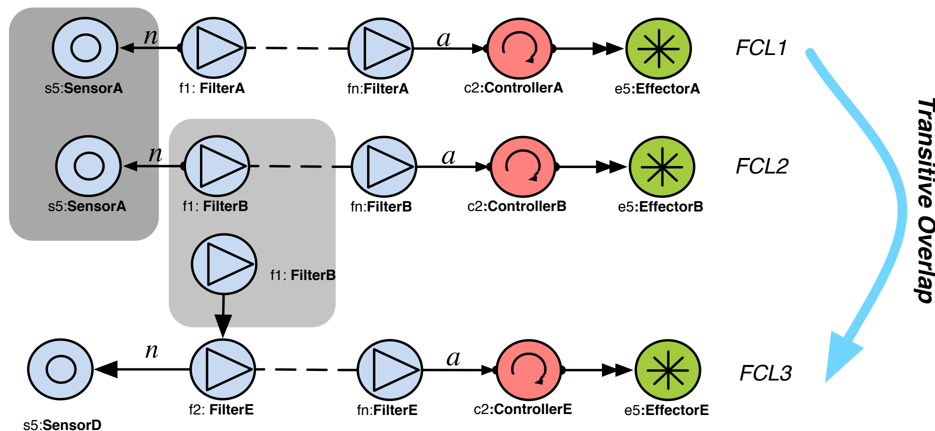

Figure 6.11: Transitive overlap Example

In general, *indirect overlaps* are the most difficult to detect in the control system architecture. For the time being, the architecture analysis of *indirect overlaps* is restrained to the scope of *transitive overlaps*. Figure 6.11 illustrates a transitive overlap between the feedback control loops, *FCL1* and *FCL3*. The figure shows that *FCL1* and *FCL2* overlap on sensor *s5*, and *FCL2* and *FCL3* overlap on filter *f1*. Since, filter *f1* depends on information provided by sensor *s5*, there is an indirect dependency between feedback loops *FCL1* and *FCL3*. The behavior of *FCL1* impacts the behavior of *FCL3*, because the decision on both feedback control loops is based on the same subset of information coming from the sensor of type *SensorA*. In

particular, when the feedback from the monitoring of data provided by the sensors does not conform to controller policies, the latter takes actions to remediate from the situation. When they are not coordinated, these actions can possibly generate conflicts in the control system.

In this section, we have presented and characterized some conflict patterns that can be a threat for the control system. The identification of these patterns enables to build algorithms for checking them in the control system architecture. In addition, it is possible to automatically generate some corrections for resolving these conflicts in the control system architecture. In the next section (cf. Section 6.4.3), we first present algorithms for the detection of *direct* and *indirect* overlaps in the control system architecture, then the mechanism for resolving detected conflicts automatically.

6.4.3 Conflicts Verification Algorithms

Conflicts verification consists in the analysis of the control system architecture for detecting potential conflicting patterns. The purpose of this verification is to enable a consistent implementation of the control system. Basically, two types of checking can be performed on the architecture model of the control system: *Direct checking* and *indirect checking*.

The *direct checking* analyses the control system architecture for detecting *direct overlap patterns*. In the same way, *indirect checking* parses the architecture model to check for *transitive overlaps*. *Transitive overlaps* are a subset of *indirect overlaps*. Below, we specify both analyses based on the following notations:

- $\mathcal{CS} = \{l_1, l_2, ..., l_k\}$ denotes the set of all loops of the control architecture model;
- $\mathcal{S} = \{s_1, s_2, ..., s_n\}$ denotes the set of all sensors of the control system;
- $\mathcal{F} = \{f_1, f_2, ..., f_m\}$ denotes the set of all filters of the control system;
- $\mathcal{C} = \{c_1, c_2, ..., c_p\}$ denotes the set of all controllers of the control system;
- $\mathcal{A} = \{a_1, a_2, ..., a_t\}$ denotes the set of all effectors (actuators) of the control system;
- $SL = \{(s_i, l_j) \in \mathcal{S} \times \mathcal{CS} : s_i$ belongs to $l_j\}$ denotes a set of relationships between sensors and feedback loops;
- $FL = \{(f_i, l_j) \in \mathcal{F} \times \mathcal{CS} : f_i$ belongs to $l_j\}$ denotes a set of relationships between filters and feedback loops;
- $CL = \{(c_i, l_j) \in \mathcal{C} \times \mathcal{CS} : c_i$ belongs to $l_j\}$ denotes a set of relationships between controllers and feedback loops;
- $AL = \{(a_i, l_j) \in \mathcal{A} \times \mathcal{CS} : a_i$ belongs to $l_j\}$ denotes a set of relationships between effectors and feedback loops.

Direct Checking

Direct checking analysis iteratively checks the control architecture model for the four types of *direct overlaps*, and namely: *sensor, filter, controller* and *effector overlaps*.

Analysis Name: *Direct checking–sensor overlap*

Algorithm: Algorithm 2 summarizes *sensor overlaps* analysis in the control system architecture. It takes as an input the control architecture CS and the set of sensors S. It produces the set of sensor overlaps detected in the control system architecture model. The algorithm *(1)* builds the relationship set SL between sensors and feedback control loops (lines 1–7), then *(2)* iterates over this set to calculate the pairs of feedback control loops that share the same sensor element (lines 8–10).

Analysis Name: *Direct checking–filter overlap*

Algorithm: Similarly to the sensor overlap checking, the *filter overlap* algorithm takes as an input the control system architecture CS and the filter set F. The algorithm produces as output the set of filter overlaps F_{ov}. The algorithm consists of two steps: In the first step, the algorithm builds the set FL, and in the second step iterates over FL to calculate filter overlaps F_{ov}. F_{ov} defines a set of triplets $< f, l_a, l_b >$. A triplet indicates that feedback control loops l_a and l_b overlap at filter f.

Analysis Name: *Direct checking–controller overlap*

Algorithm: The *controller overlap* checking algorithm takes as an input the control architecture CS and the controller set C in order to calculate the set of controller overlaps C_{ov}. The algorithm processes the control system architecture in two steps: In the first step the algorithm builds the set CL of relationships between controllers and feedback control loops. The second step consists of iterating over CL in order to calculate C_{ov}. C_{ov} defines a set of triplets $< c, l_a, l_b >$. A triplet indicates that a controller c is the point of overlap between feedback loops l_a and l_b.

Analysis Name: *Direct checking–effector overlap*

Algorithm: The *effector overlap* checking algorithm takes as input the control architecture model CS and the effector set A, and produces the set of effector overlaps A_{ov}. The algorithm processes the control system architecture in two steps: In the first step, the algorithm builds the set AL of relationships between effectors and feedback loops. In the second step, the algorithm iterates over AL in order to derive A_{ov}. A_{ov} defines a set of triplets $< a, l_a, l_b >$. A triplet indicates that an actuator a is the point of overlap between feedback loops l_a and l_b.

Algorithm 2 Direct Checking: Sensor Overlap

Require: Set $\mathcal{CS}$ of control loops, Set S of sensors
Ensure: Calculate the Set S_{ov} of sensor overlaps

1: **for all** $s \in S$ **do**
2:   **for all** $l \in CS$ **do**
3:     **if** $s \in l$ **then**
4:       $SL \leftarrow SL \cup \{s, l\}$
5:     **end if**
6:   **end for**
7: **end for**
8: **for all** $(sl_i, sl_j) \in SL^2$ such that $sl_i \cap sl_j = \{s\}$ **do**
9:   $S_{ov} \leftarrow S_{ov} \cup \{s, l_i, l_j\}$
10: **end for**

Indirect Checking

For the time being, *indirect checking* in *CORONA* is limited to the scope of *transitive overlaps*. *Transitive overlaps* algorithms check the control system architecture for detecting indirect dependency between feedback control loops that share a subset of information, but implement different control policies. *Transitive overlap* can be a threat for the stability of the control system because of the invisible interference that exist between feedback control loops.

Algorithm 3 Indirect Checking: Transitive Overlap

Require: Set $\mathcal{CS}$ of control loops, Set S_{ov} of sensors overlaps, Set F_{ov} of filters overlaps
Ensure: Calculate the Set TR_{ov} of transitive overlaps

1: **for all** $l \in CS$ **do**
2:   **for all** $< s, l_a, l_b > \in S_{ov}$ **do**
3:     **if** $< s, l_a, l_b > \cap \{l\} = \emptyset$ **then**
4:       **for all** $< f, l_n, l_m > \in F_{ov}$ and $< f, l_n, l_m > \cap \{l\} = \{l\}$ **do**
5:         **if** $< s, l_a, l_b > \cap < f, l_n, l_m > \neq \emptyset$ **then**
6:           $TR_{ov} \leftarrow TR_{ov} \cup \{< f, l_n, l_m >, < s, l_a, l_b >\}$
7:         **end if**
8:       **end for**
9:     **end if**
10:   **end for**
11: **end for**

Algorithm 3 summarizes the main steps of the *transitive overlap* checking. The algorithm takes as an input the control system set CS, the set of sensor overlaps S_{ov}, the set of filter overlaps F_{ov}, and produces as output the set TR_{ov} of transitive overlaps. The algorithm consists of two steps: On the first step, the algorithm select a set of three loops (*lines 1–3*) for which the first two overlap on a sensor. On the second step, the algorithm checks wether there exists or not a filter overlap (lines 4–8) between the loops selected on the previous step. The set of transitive overlap is progressively built by iterating over all the loops of the

system.

6.4.4 Conflicts Resolution

When conflicts are detected during the architecture analysis, the architect receives warnings about detected conflicts. At this stage, conflicts can be resolved in two ways: manually or automatically.

Manual conflict resolution is performed by the architect. Upon the reception of warning notifications by the analysis tool, the architect can redesign the control architecture in order to take into account these notifications. He/She can also decide to ignore these notifications. In this case, he/she should change the status of the warning notification into *resolve or cancel*. This is to disable the automatic conflict resolution by the *CORONA* toolchain.

Automatic conflict resolution is performed by the *CORONA* toolchain, when the control system architecture is provided to the code generator with unresolved warnings. In this context, the *CORONA* toolchain tries to resolve these warnings automatically. The *CORONA* toolchain implements two types of mechanisms for resolving conflict warnings: The *proxy resolution pattern*, and *supervisors mechanism*.

A) **Proxy Resolution Pattern**
The *proxy resolution pattern* is a mechanism implemented by the *CORONA* toolchain in order to coordinate the behavior of conflicting feedback control loops. The principle behind the *proxy pattern* is to provide a *coordination node* between conflicting loops.

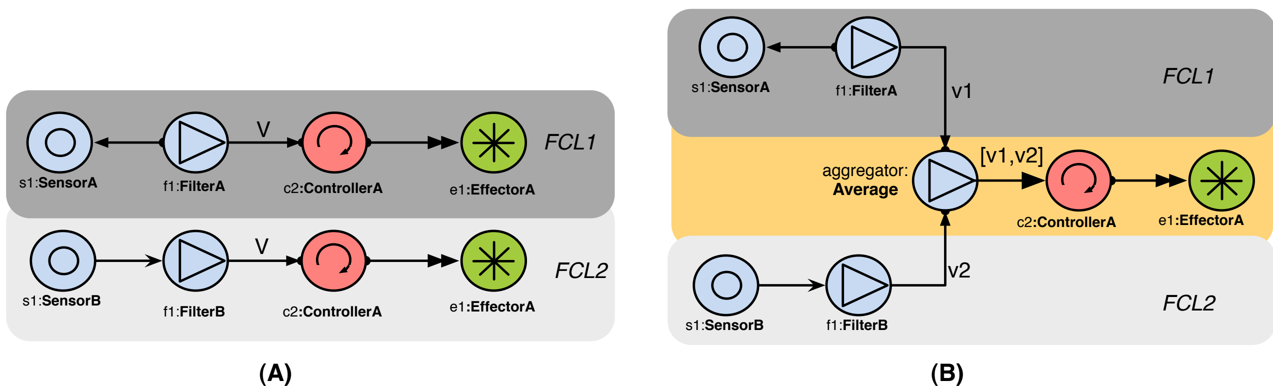

Figure 6.12: (A)– Control Architecture with Conflicts, (B)– Resolution of Architecture Conflicts with Proxy Pattern

To illustrate the *proxy resolution pattern*, let us consider a control system architecture that consists of two feedback control loops with an *effector* and a *controller* overlap. An example of such control system architecture is depicted on Figure 6.12–A. The resolution of conflicts detected on the control architecture is depicted on Figure 6.12–B. The resolution pattern applied

here is the *proxy resolution pattern*. Figure 6.12–B shows that the new architecture obtained after conflict resolution contains the coordination node *aggregator*, which is a filter control element. The aggregator element computes the average of incoming values v_1 and v_2. The type of the computation–(calculation of the average)– can be customized during the implementation phase. The coordination node *aggregator* enables the coordination of the feedback loop *FCL1* and *FCL2*.

The *proxy resolution pattern* tries to provide a proxy element between conflicting loops. The type of input/output values is inferred from the architecture analysis. Typically, on Figure 6.12–A, the output (v) from filters are used as input for the proxy element. Unfortunately, it is not always possible to resolve control system architecture conflicts with the proxy resolution pattern. When the *CORONA* toolchain cannot resolve conflicts through one resolution mechanism, it tries to apply other available mechanisms like the supervisor mechanism.

B) **Supervisor Mechanism**

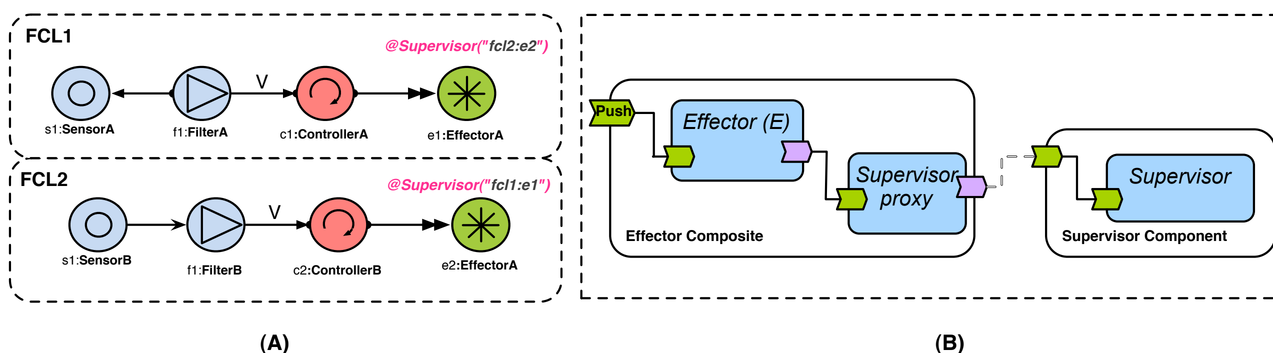

Figure 6.13: (A)– Resolution of Conflicts with the Supervisor Mechanism, (B)– SCA implementation of the Supervisor Mechanism on an Effector

The *supervisor mechanism* is a mechanism of conflict resolution used by the *CORONA* toolchain. The purpose of the *supervisor mechanism* is to generate *supervisors* for the control architecture in order to coordinate the behavior of conflicting feedback control loops. *Supervisors* are a specific type of controllers (processor), which implement a coordination mechanism. The coordination logic of *supervisors* operates on the basis of *effector overlaps*. The main purpose of the *supervisor mechanism* is to avoid concurrent actions of conflicting feedback control loops at run time.

If we slightly change Figure 6.12–A, by removing *controller overlaps*, the proxy resolution pattern will fail. Therefore the *CORONA* toolchain will try to execute the observer mechanism for resolving *effector overlaps*. Figure 6.13-A gives an illustration of control architecture with *effector overlaps*, for which *supervisor mechanism* is used for resolving conflicts. Figure 6.13-A depicts a control system architecture where effectors are decorated with

annotations–*@Supervisor("param")*. The *@Supervisor* annotation indicates for each effector the list of influencing effectors. On the figure, effector *e1* of *FCL1* is influenced by effector *e2* of *FCL2*. The *@Supervisor* annotation is later instrumented by the code source generator, for generating observers components.

To illustrate the importance of the *supervisor mechanism*, it can be interesting to exemplify the control system architecture presented on Figure 6.13–A, with a concrete illustration. Therefore, let us suppose that the objective of the *FCL1* loop is to control the availability of resources on a web server. Similarly, the objective of *FCL2* is to repair resources when they failed. When a resource fails, *FCL1* detects it, allocates a new resource and moves files from the failed resource to the new resource. Since both control loops operate on the same resources (*effector overlap*), a conflict for accessing resources can happened when *FCL1* tries to move files from the failed resource, and *FCL2* to get access to that resource for reparation. In this context, both feedback control loops need for a coordination and this need can be fill by the *supervisor mechanism*.

Figure 6.13–B depicts the implementation of the supervisor mechanism in SCA. The supervisor mechanism uses the push/subscribe pattern for coordinating the behavior of conflicting loops. Each effector involved in the conflict encapsulates an *supervisor-proxy* component. The *supervisor-proxy* component enables the communication with the *supervisor* component. The *supervisor* component plays the role of a registry. Upon call of the *push* service, the effector composite query for the status of conflicting effectors from the *supervisor*. The coordination logic of the supervisor mechanism is illustrated on the sequence diagram depicted on Figure 6.14.

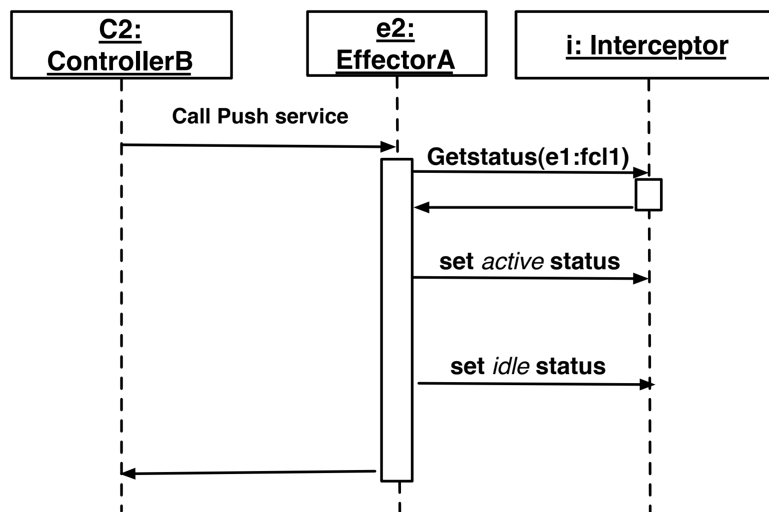

Figure 6.14: Supervisor Mechanism Coordination Logic

On Figure 6.14 the sequence of coordination consists of the following steps:

1. Call of a method *m* provided by the *push* service of the effector *e2*.

2. Collect the status of influencing effectors. The list of influencing effector is calculated during the architecture analysis, and provided for the runtime implementation through the annotation *@Supervisor*. When influencing effectors have the *idle* status, the execution of the method *m* can proceed.

3. Change the status of effector *e2* to *active*, and publish it for the supervisor.

4. Change the status of effector *e2* to *idle* at the end of the execution of method *m*. Then publish the new status for the supervisor.

In this section we have presented two mechanisms for resolving conflicts in the control system architecture. In the next Section 6.5, we discuss how the evolution of the control loop architecture is supported in *CORONA*.

6.5 Control Loop Architecture Evolution

Architecture evolution is an important aspect of the process of implementing feedback control loops. The term *evolution* refers to the possibility that is given to developers to modify or to change the control system architecture without lost of the part of implemented logic behavior. Figure 6.15 gives an overview of the three phases of the control system architecture evolution.

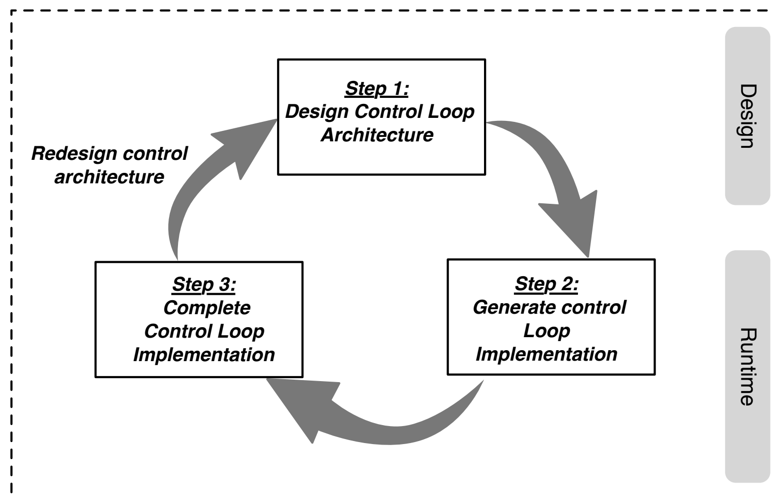

Figure 6.15: Control Loop Architecture Evolution Cycle

The *first phase* (cf. Step 1–Figure 6.15), is the design of the initial control system.

The *second phase* (cf. Step 2–Figure 6.15), consists in the generation of the control system architecture.

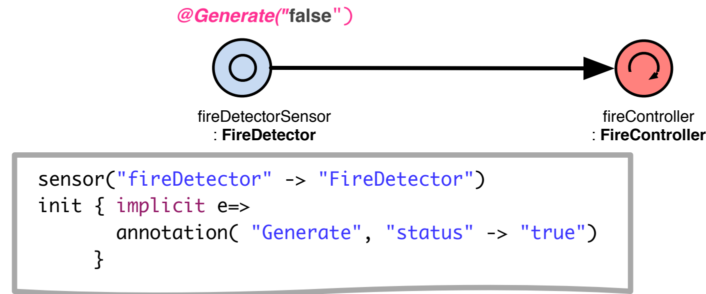

Figure 6.16: Control Architecture Selective Generation

The *third phase* (cf. Step 3–Figure 6.15), consists in the completion of the control system implementation code.

After the *third phase*, the developer can modify the control system architecture according to his/her needs, and go again through the evolution process from the *first phase*. In order to avoid any lost of the part of the code added by the developer, *CORONA* enables a selective generation of the implementation code. This is possible by annotating the control system architecture model.

Figure 6.16 depicts how the architecture of the control system can be annotated in order to trigger a selective generation of the control system implementation code. On the Figure, the sensor *fireDetectorSensor* is annotated with the annotation *@Generated("false")*. The annotation *@Generated* with parameter *false*, indicates to the code source generator that an architecture elements does not need to be generated.

6.6 Summary

In this chapter we have presented the *CORONA* toolchain. We have given the focus on tools support provided for developers of autonomic systems. In particular, we have highlighted the code source generation process and architecture verifications. We have identified conflict patterns on the control architecture, and suggested mechanisms for detecting and resolving them. We have concluded this chapter by addressing the evolution of the control system architecture in our approach.

This chapter concludes the contribution part of this dissertation. In Chapter 4 we gave a glimpse of the *CORONA* approach as well as challenges that the current thesis aim to address. We stated that the originality of our approach revolves in increasing the visibility of feedback control loops at runtime, while enhancing developers experience when engineering autonomic systems. In Chapter 5, we dived into details by presenting the runtime architecture of feedback control loops in our approach. We explained how control elements

were reify as first-class entity at runtime. We also described how the MAPE-k architecture model can be customized with cross-cutting concerns. In particular we discussed how stabilization mechanisms can help implementing stable control system. Chapter 6 focused on the compilation infrastructure of the *CORONA* toolchain. The next part of this dissertation discusses the benefits and limitations of the *CORONA* approach through experiments on three case studies.

This page was intentionally left blank

Part III

Validation

Chapter 7

Condor Case-Study

"The greatest challenge to any thinker is stating the problem in a way that will allow a solution."–
Bertrand Russell

Contents

7.1 Case-study Objective **112**
7.2 Condor Case-Study Description **112**
7.3 Control System Architecture **113**
7.4 Quantitative Evaluation **115**
7.4.1 Characterization of Conflicts Detection Algorithms 115
7.4.2 Condor Evaluation Test bed 116
7.4.3 Characterization of the Condor's Control system 117
7.5 Summary **120**

This chapter is the first of the three chapters of the validation part. The validation part is organized as following: The first chapter (cf. chapter 7) illustrates the consistency of control architecture implemented with the *CORONA* approach. It exemplifies how conflict detection algorithms can be used for implementing consistent, conflict-free control architecture. The second chapter (cf. Chapter 8) illustrates the *integration & transparency* objective of the *CORONA* approach. Finally, the third chapter (cf. Chapter 9) demonstrates how stabilization mechanisms can be used for implementing stable control systems.

This chapter is organized as follows: Section 7.1 presents the objective of the condor case-study. The description of the scenario is discussed in Section 7.2. Section 7.3 explores the control system architecture corresponding to the case-study. The results of the quantitative evaluation of the condor case-study are elaborated throughout Section 7.4. Finally, we conclude (Section 7.5) this chapter with a summary.

7.1 Case-study Objective

Consistency refers to building reliable autonomic applications. In chapter 6, we have identified that, one of the threat for building large-scale autonomic systems is the complexity of coping with conflicts in the control system architecture. In our approach, we have implemented a set of algorithms for helping developers to build consistent autonomic systems. It is important to notice that the *CORONA* toolchain checks the control system architecture only against referenced conflict patterns identified in Chapter 6.

The implementation of automated conflicts checking in the control architecture enhances the work of developers of autonomic systems. It also reduces the cost related to the maintenance of these systems. In the next section, we illustrate how the *CORONA* approach helps developers for handling conflicts of the control system. In particular, this example focus on effector and controller overlap patterns.

7.2 Condor Case-Study Description

In this section, we first describe the Condor scenario example. Then, from there, we identify conflicting policies of the control system.

Scenario Overview
The Condor scenario is a validation case-study of the SALTY project. We consider a cloud environment for executing intensive workflows for a company. This executing environment is distributed and is provided with a scheduler that is responsible for managing submitted jobs and mapping them onto a set of resources where the actual execution is performed. The *space-shared* policy is used for allocating jobs to resources. Since workflows tend to be rather large, containing many computer-intensive tasks, the scheduler can easily become overloaded, as the more tasks it has to handle the more resources it uses. The default behavior of the scheduler is to accept all valid submission requests regardless of the current state of the system.

We consider a family of legacy applications deployed by a company in the Condor environment. Each of these applications are configured to ensure a specific Service Level Agreement (SLA) to their users. To continuously maintain this SLA, each application exploits the elasticity of the distributed environment to provision more or less computing resources according to incoming requests. The usage cost associated to the provisioned resources is generally computed as the product of the *resource cost per hour* (CostRes/hour) and the duration of the lease (LeaseRes). To keep being profitable, the company owning applications has defined a maximum budget per month for provisioning resources when needed (BudgetMax/month). Therefore, in order to save money, the control system tries to release provisioned resources as soon as possible considering constraints on the budget, and on the SLA for users.

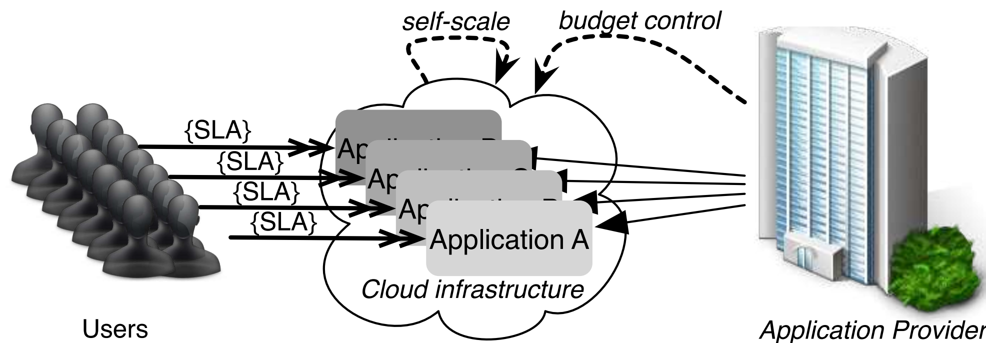

Figure 7.1: Self-Adaptive Distributed Infrastructure

Figure 7.1 illustrates the Condor scenario. Users request services provided by applications deployed in the Condor environment, while an autonomic manager takes care of monitoring and maintaining SLA for each user. Resources are provisioned and released by the dedicated manager hosted by the Cloud infrastructure. This manager implements two feedback loops: The *self-scale* feedback control loop that ensures the respect of the user SLA, and the *budget* feedback control loop that controls the budget allocated for the leasing of the platform.

Architecture Conflits

In this scenario, the autonomic manager executes two conflicting policies: (*i*) The first policy consists of maintaining an acceptable SLA for users, which is mostly achieved by provisioning new resources. (*ii*) The second policy consists of saving the company provisioning budget, and this is achieved by reducing the number of provisioned resources. To guarantee a predictable behavior of deployed applications, the autonomic manager must be aware of these conflicting goals (*i*) and (*ii*), and should encapsulate strategies to handle these conflicts.

7.3 Control System Architecture

According to the scenario, the control system architecture consists of two feedback control loops: the self-scale and the budget feedback control loop. From the scenario, we derive the specification of adaptation policies which are required to continuously adapt the system. The specification of these policies is based on the SALTY model. The SALTY model fosters the visibility of the control system architecture by reifying their constituents as first-class elements. The *self-scale* and the *budget* feedback control loops that manage the Condor platform are depicted on Figure 7.2 and Figure 7.3, respectively.

The self-scale or auto-scale feedback control loop periodically monitors incoming requests rate (μ), and computes an average value using a sliding window ($\overline{\mu}$), as reported in

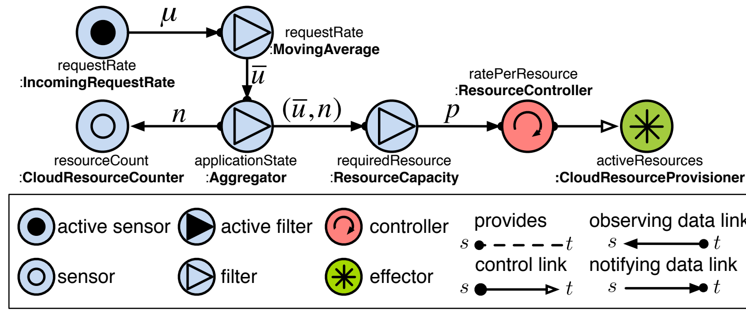

Figure 7.2: Self-Scale Feedback Loop

Figure 7.2. The computed average value is aggregated with the current number of provisioned resources (n), and used for evaluating requirement for resource allocation (p). Resource allocation requirement is used by the resource controller to provision the appropriate number of resources when it differs from the current number of allocated resources.

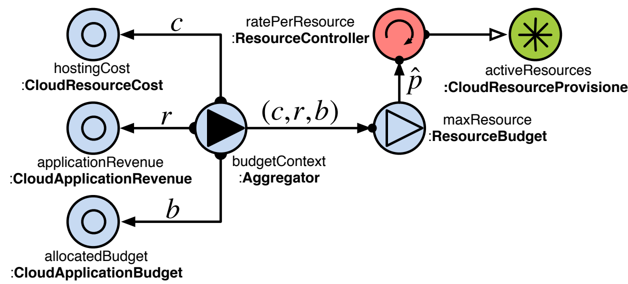

Figure 7.3: Budget Feedback Loop

The budget feedback control loop aggregates the hosting cost (c), applications revenue (r), and allocated budget (b) to compute allowed resource budget ($\hat{p}$) as depicted on Figure 7.3. The resource budget is used to ceil the number of allocated resources in order to control the resource consumptions of deployed applications.

Incidentally, when deploying both feedback control loops concurrently, the behavior of managed applications becomes unstable. In particular, one can observe that, when reaching the maximum budget under increasing request rates, both feedback control loops tend to take conflicting decisions with regards to the number of resources to be provisioned, thus making the whole system unstable. Furthermore, the probability of facing conflicting decisions increases with the number of concurrent adaptation policies, thus making a manual detection difficult. *CORONA* provides support for analyzing the control system architecture against conflictual policies. This help developers of autonomic systems to implement more reliable applications. In the next section, we provide a quantitative evaluation that underlines the importance of detecting and managing conflicting policies in autonomic sys-

tems. In particular, we evaluate the behavior of the control system before and after conflict resolution.

7.4 Quantitative Evaluation

This section aims at demonstrating how conflicts checking enables to implement consistent control systems. We present here results from experiments with the Condor control system presented in Section 7.3. We introduce a characterization of conflicts detection algorithms implemented in *CORONA*, then we discuss the behavior of the condor control system for different architecture configurations.

This section is organized as follows: In Section 7.4.1, we provide and empirical evaluation of the time required for checking control system architectures against conflicts using our algorithms for conflicts detection. Prior to present results from the experiment, we describe in Section 7.4.2 the experiment test bed. Finally, Section 7.4.3 described the result of the experiment we conducted. In this experiment we strive to evaluate the behavior of the Condor control system with and without architecture conflicts.

7.4.1 Characterization of Conflicts Detection Algorithms

In *CORONA*, we use a control-oriented syntax based on the *MAPE-K* paradigm to reify the structure of the control flow in the control system architecture. The visibility of the control flow facilitates the analysis of the control system architecture in order to detect conflicting policies. In chapter 6, we have presented some conflicts detection algorithms implemented in *CORONA*. In order to characterize these algorithms, we have conducted some experiments. Figure 7.4 illustrates the result of these experiments.

Figure 7.4 shows the algorithm execution time variation depending on *loop complexity (K)*. The loop complexity criteria is the product of *the number of architecture elements* (*nodes* and *edges*) and *the number of overlaps* over *the number of feedback loops*. The number of overlaps corresponds to the number of known conflicts pattern between the feedback control loops of the control architecture. The figure shows two curves, the curve with the diamond line points shows variations of the algorithm execution time for 3 overlaps in the control system architecture. Respectively, the square line points shows variations for 10 overlaps. All points correspond to the average value, obtained after executing the algorithm on the architecture 100 times. The figure shows that for 200 architectural elements (complexity of 1.2), overlaps are found in less that *16ms*[3]. We also notice that the behavior of the algorithm seldom varies when the number of overlaps in the architecture triples (*from 3 to 10 overlaps*). This is confirmed by the fact that the algorithm execution time remains moderate with an important number of overlaps in the architecture. For example, for 200 architectural elements and 50 overlaps in the architecture, we notice that the execution time is around *250ms*.

[3]notice that, most of the existing control system can be designed with very less elements.

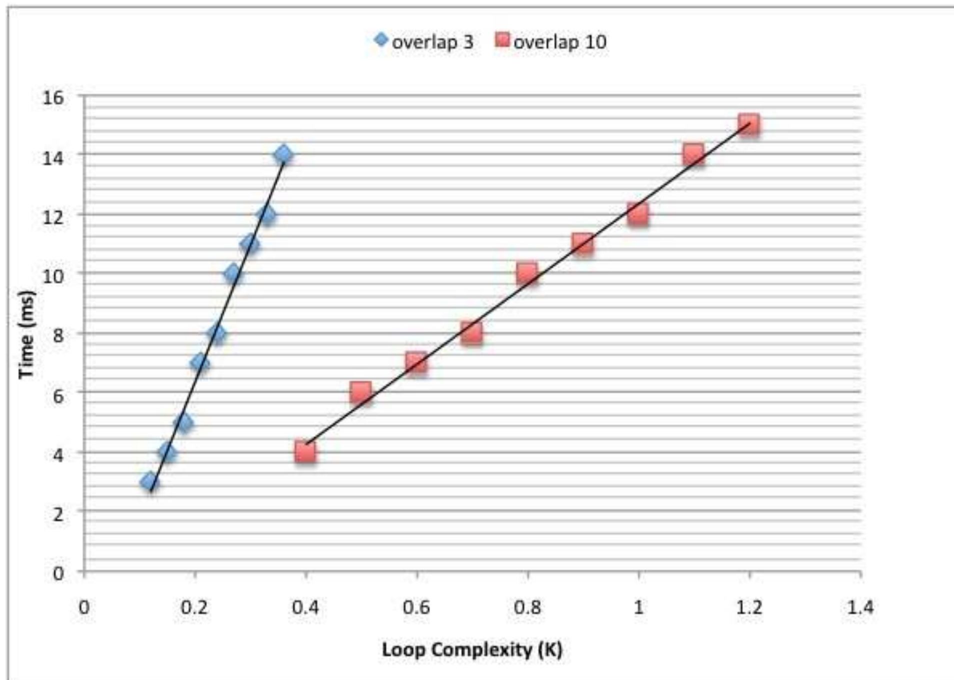

Figure 7.4: Feedback Loops Overlaps Detection Time

The fact that the execution time of the algorithm is low is an essential criteria, especially for developers of autonomic systems. This is because it will significantly reduce the time they spend for checking conflicts in the architecture. We agree that an algorithm faster than ours may exist, but since our algorithm takes less than $1s$ to be executed, developers will hardly notice the difference between both algorithms.

The experiment was conducted on a *Intel core 2 duo 2.8Ghz* processor. The algorithmic precision for detecting *direct overlaps*, and *transitive indirect overlaps* is of 100% for all the tests we performed. The detection of other types of *indirect overlaps* is not trivial, but the job mainly consists in identifying a pattern for each of them, as we have showed for *transitive overlaps*. Later, theses patterns can be used to formalize and implement algorithms for the detection of these overlaps in the architecture.

7.4.2 Condor Evaluation Test bed

GridSim [BM02] is an open source library that allows modeling and simulation of entities in parallel and distributed computing (PDC) systems-users. We setup the following experiment based on *GridSim*, to evaluate how the conflicting policies mentioned above can affect the behavior of the deployed application. The testing environment is an Intel core 2 duo

2.8Ghz on which *GridSim* is installed. The *GridSim* environment consists of a initial set of 9 resources, consisting of *Sun Ultra* processing element (PE) architecture, and initial set of 50 clients sending jobs according to a poisonous distribution law. We used a space-shared algorithm for jobs scheduling. The network topology is the same as the one depicted on Figure 7.1. The bandwidth between users/company and the cloud infrastructure is set to 500MHz and within the cloud infrastructure entities (scheduler, queue, etc...) to 1GHz.

7.4.3 Characterization of the Condor's Control system

To characterize the Condor control system, we have performed some experiments. Figures 7.5, 7.6, 7.7, 7.8 present the results that we have obtained for different configurations.

Figure 7.5 depicts the behavior of the condor application without any control system. Figure 7.6 and Figure 7.7 describes the behavior of the system with a single feedback control loop, the budget FCL and the autoscale FCL respectively. Figure 7.8 depicts the system behavior when both feedback control loops (budget and autoscale) are running without coordination. In addition to these figures, we described through Figure 7.10 the behavior of the condor system when both feedback control loops are coordinated thanks to the detection of conflicts through our checking algorithms.

On each of the figures, the curve (A) shows the variation of provisioning resources, curve *B* the workload of the system, curve *C* the variation of the budget, curve *D* the CPU usage, curve *E* the amount of running jobs, curve *Fb* the budget threshold and curve *Fc* the CPU threshold.

Figure 7.5 depicts the behavior of the standalone application behavior without the autonomic manager. We can observe that the *cpu-usage* is quickly saturated to reach 100% (maximum available). The same observation is valid for the budget variation that grows rapidly. Figure 7.6 and Figure 7.7 show the behavior of the system with a single feedback loop. On Figure 7.6, we can observe that the budget curve *C* remains below the budget-threshold curve *Fb*. This is because, the budget feedback control loop free provisioned resources to maintain the cost of resources utilization affordable for the company. The curve *A* shows that the amount of available resources decreases.

Similarly, Figure 7.7 shows that the variation of the cpu-usage on curve *D* does not go beyond the cpu-usage threshold curve *Fc*. This is because, the autonomic manager allocates new resources to maintain the user's SLA acceptable. Meanwhile, on that figure, the curve *C* of the budget is above the budget-threshold. Finally, Figure 7.8 depicts the behavior of the system with an autonomic manager which consist of uncoordinated budget and SLA feedback control loops. The global behavior of the system observed on Figure 7.8 is not better than the one obtained with the standalone application on Figure 7.5. On Figure 7.8, the curve *A* of the resource variation indicates the instability of the system because resources are being allocated and de-allocated continuously.

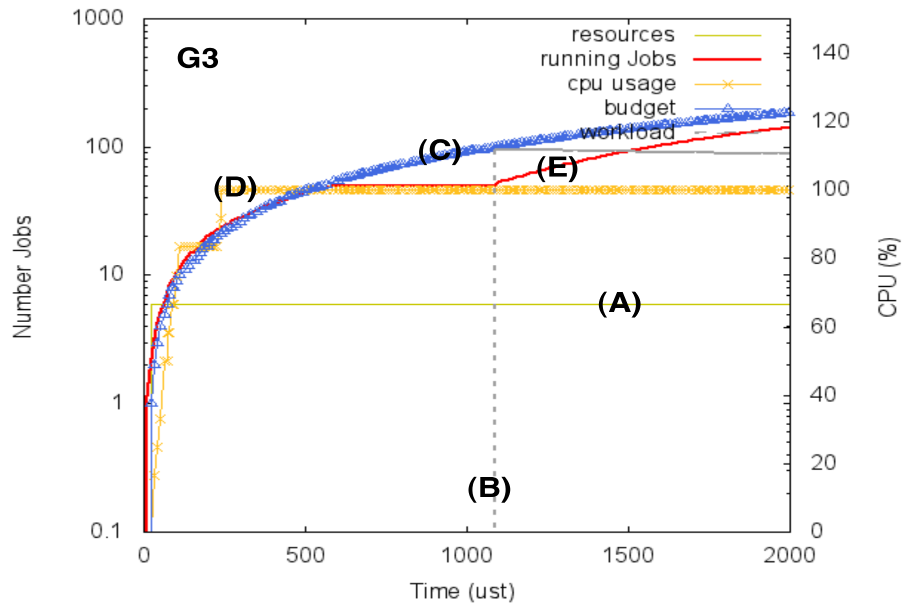

Figure 7.5: System without Feedback Loops

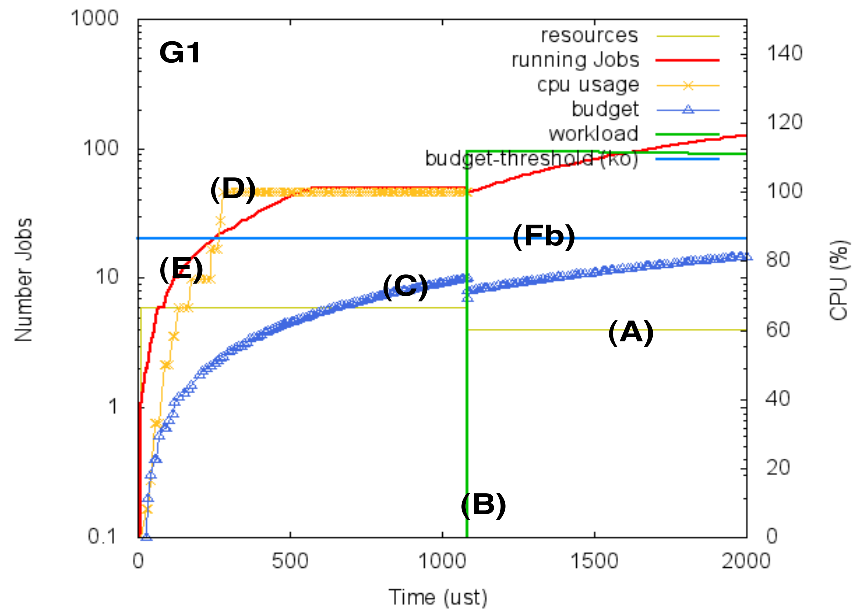

Figure 7.6: System With a Single Budget Feedback Loop

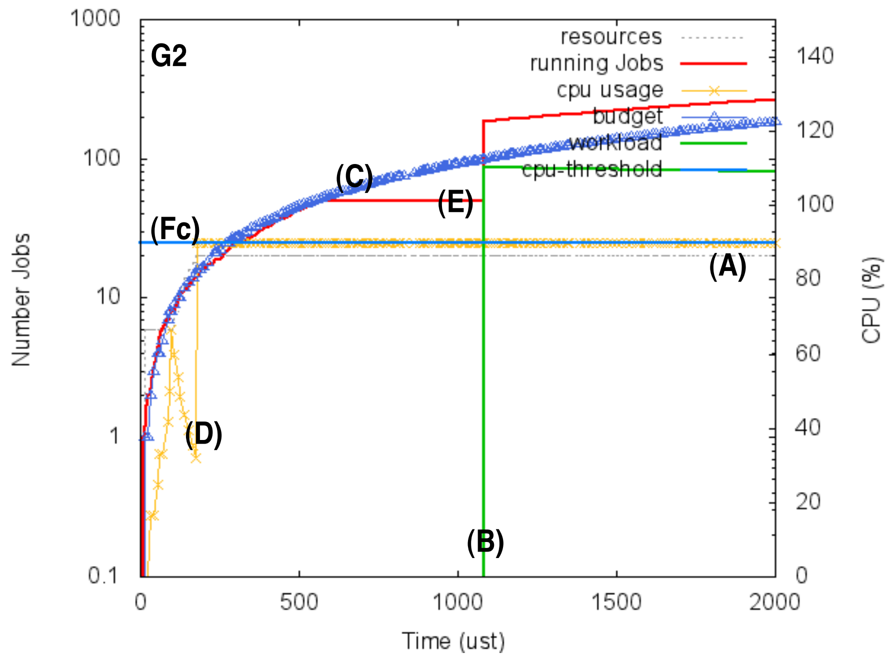

Figure 7.7: System With a Single User SLA Feedback loop

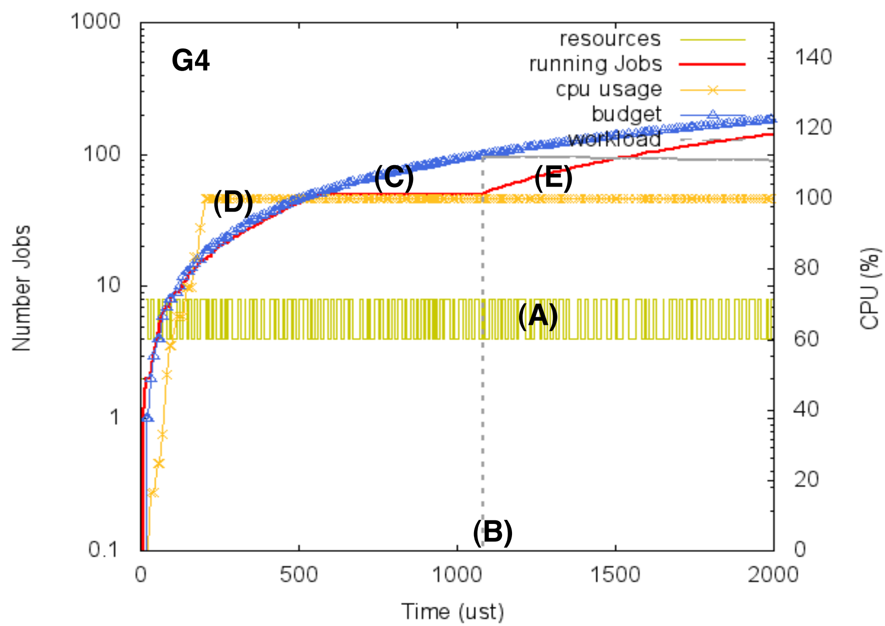

Figure 7.8: System with Uncoordinated Feedback Loops

These results underline two important facts: First, the *need* of coordination strategies between conflicting feedback control loop policies of the autonomic manager. Second, the *importance* to evaluate the behavior of an autonomic manager before the deployment on a real infrastructure. By applying the *CORONA* approach for detecting conflicting policies in the case of the Condor use case, we obtain a good coordination of the control system. Figure 7.10 depicts the behavior of the autonomic application when feedback loops are coordinated according to the control system architecture provided on Figure 7.9.

Figure 7.10 consists of several annotated curves: curve *A* correspond to resources variation for the experiment set, curve *B* shows the workload variation, curve *C* the variation of the budget, curve *D* the CPU usage, curve *E* the number of running jobs, curve *Fc* the CPU threshold and curve *Fb* the budget threshold. By analyzing these curves, we observe that the CPU usage curve *D* does not go above the CPU threshold curve *Fc*. Similarly, the budget curve *C* remains below the budget threshold *Fb*. From these observations, we can conclude that the implemented conciliation strategy of the budget and users SLA conflicting was effectively resolved in this experiment. However, it is not always the case, for example we realized that if the allocated budget per month is reduced by 10%, users SLA policy will not be guaranteed. That is because, the *double Threshold* [NRS10] coordination strategy gives priority to budget over the users SLA.

7.5 Summary

The prevention of FCL side effects is an important issue for autonomic systems in order to make them reliable, especially in the context of a multi-objectives control systems. In this chapter, we have presented some experimental results to asses the consistency claim of the *CORONA* approach. We have used the Condor case-study, to explain architecture conflicts in autonomic systems. Then, we have presented how the coordination between feedback loops can help to achieve better results. In the next chapter, we show how the reusability and the heterogeneity properties are supported in the *CORONA* approach. We illustrate these properties through the fire-emergency case-study.

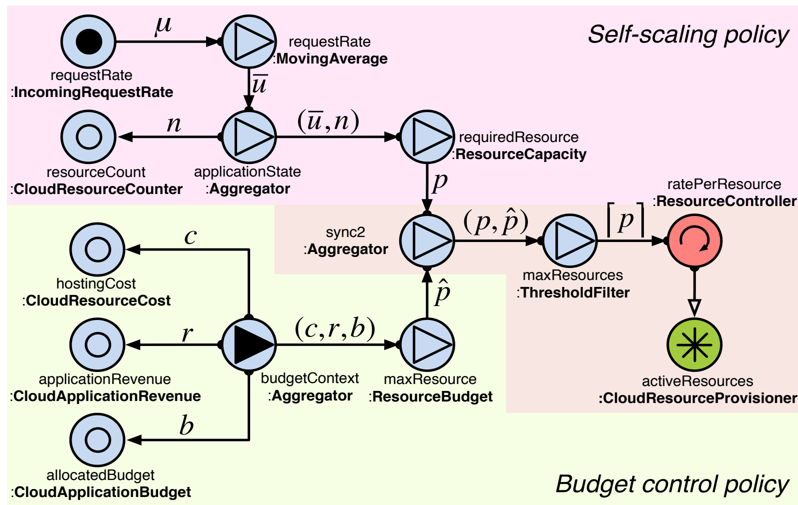

Figure 7.9: coordinated Control System Architecture

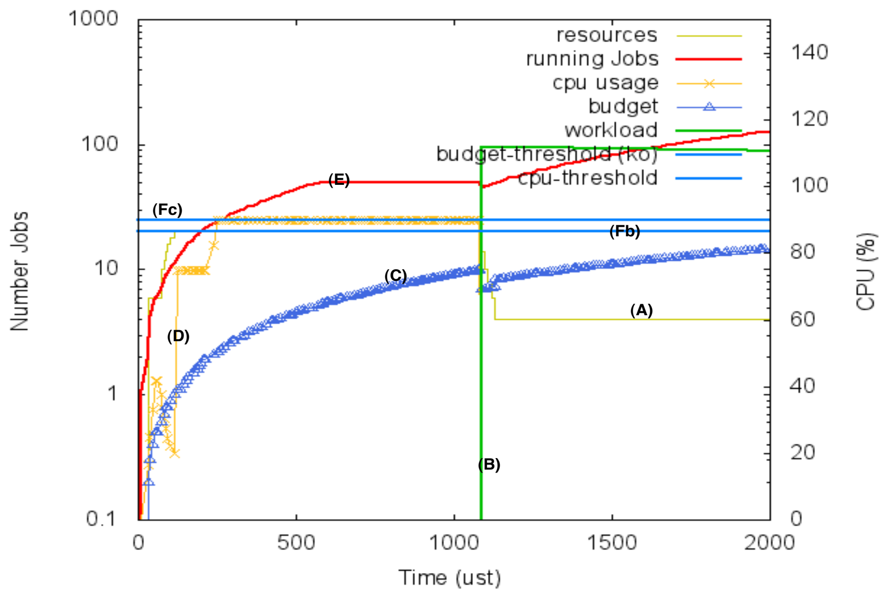

Figure 7.10: System with Coordinated Feedback Loops

This page was intentionally left blank

Chapter 8

Fire Emergency Case-Study

"We should be taught not to wait for inspiration to start a thing. Action always generates inspiration. Inspiration seldom generates action."
–Frank Tibolt

Contents

8.1 Case-Study Objective 124
8.2 Scenario Description 124
8.3 Control System Architecture 125
8.4 Control System Implementation & Measures 126
8.5 Summary 130

In the previous chapter, we have demonstrated with the Condor case-study, the importance of providing support for conflicts checking analysis when building autonomic systems. In particular, we have shown that automatic architecture verification helps developers of autonomic systems to build more reliable applications. As a consequence, it also reduces the maintenance costs of autonomic systems. In this chapter, we demonstrate how *CORONA* can be used for enhancing existing software systems with autonomic capabilities. The main goal of this chapter, is to show that feedback control loops implemented through the *CORONA* approach can be integrated effortlessly within existing systems. The *generality* of *CORONA* is illustrated through the Fire-emergency case-study.

Structure of the Chapter

This chapter is organized as follows: We start by recalling some issues related to the engineering of autonomic systems that justify requirements for a generic engineering approach (cf. Section 8.1). In Section 8.2, we describe the Fire-emergency scenario. Then, in Section 8.3, we present the control system architecture of the Fire-emergency case-study. In Section 8.4, we introduce the runtime implementation architecture of the control system generated by the *CORONA* toolchain, and discuss some measures related to engineering efforts required for implementing this architecture. We conclude this chapter by a summary (cf. Section 8.5).

8.1 Case-Study Objective

In the first chapter of this thesis, we have explained that autonomic systems were built from heterogeneous technology platforms, and that in order to cope with this *heterogeneity*, we needed a technology agnostic approach for building autonomic solutions. In the contribution part of this dissertation, we have shown that the *CORONA* approach is platform agnostic, and provides developers of autonomic systems a control-oriented syntax for implementing autonomic behaviors. In this chapter, we want to enforce the generality aspect of the *CORONA* approach, by presenting how it can be used for enhancing existing software with autonomic capabilities. For that purpose, we use the Fire-emergency case-study. This case-study is interesting because, it demonstrates how interoperability can be achieved between heterogenous components of the control system.

The objective of the fire-emergency case-study is twofold:

- Demonstrate the generality of the *CORONA* approach. This is realized by showing how heterogenous technology protocols can be combined for implementing an autonomic manager. This is possible thanks to the FraSCAti runtime which supports interoperability between heterogeneous components.
- Demonstrate the reusability of existing components. This objective is achieved by showing how implemented software components can be reused for implementing an autonomic manager.

8.2 Scenario Description

The Fire-emergency scenario was developed for the ITEMIS[4] ANR project. The purpose of the scenario is to build an autonomic manager for handling a fire emergency crisis in a smart environment. The smart environment is composed of sensors (*Fire-detector*) that sense the environment for detecting fire-emergency situations. This environment is also composed of actuators, and namely sprinklers and turnstile doors. In case of fire, sensors must detect it, sprinklers should be triggered and turnstile door open to allow evacuation. In addition, a notification message should be sent to a Twitter account to inform subscribers about the emergency. The current status of each actuator is registered at the fire controller center. Figure 8.1 gives an illustration of the fire-emergency scenario.

Figure 8.1 depicts the relationship between components involved in the fire-emergency scenario. Fire-detectors collect information about the environment and notify actuators components. When their status change, actuators notify the fire-controller center about that change. The fire-emergency scenario is interesting for illustrating the generality of the *CORONA* approach, because it involves heterogeneous components. In particular, SOAP,

[4] http://www.itemis-anr.org/

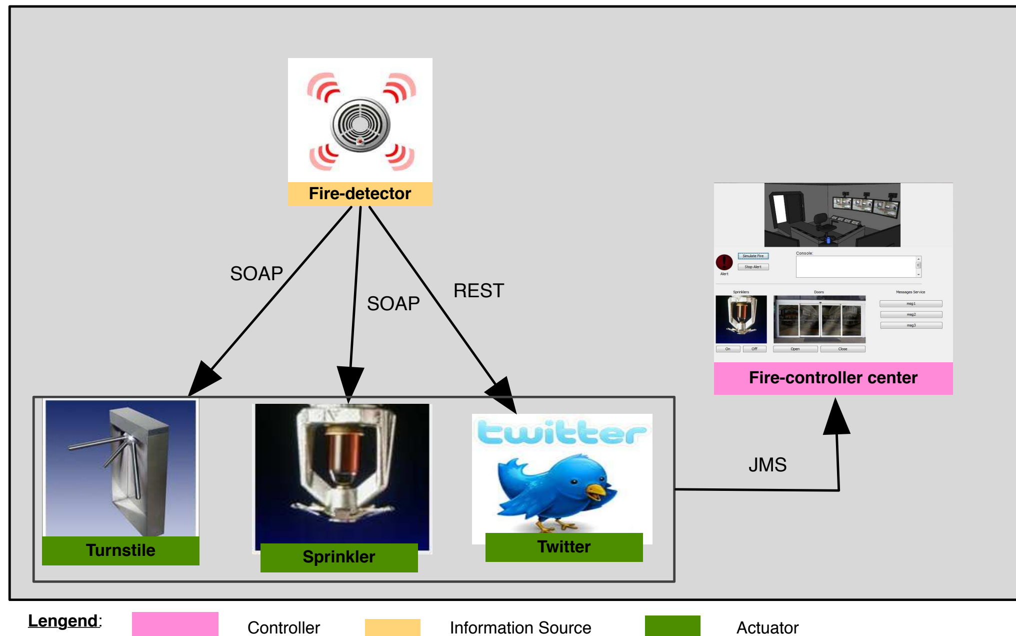

Figure 8.1: Illustration of the Fire-emergency Scenario

REST and JMS protocols are used for exchanging messages between the component of the control system. Let now have a detailed look at how this control system can be designed using *CORONA*.

8.3 Control System Architecture

In this section, we provide the architecture of the control system of the fire-emergency scenario. We put the focus on how the control system architecture can be enriched for driving the generation of its implementation code.

Figure 8.2 gives an overview of the fire-emergency control architecture. This architecture is composed of *Sensors*, *Controllers*, and *Effectors*.

- *Sensors*. Figure 8.2 depicts two types of sensors. An active sensor, *fireDetectorSensor* which sends alarm signal to the controller *fireController*; and three passive sensors *tunrstileSync*, *sprinklerSync* and *twitterSync* which are provided element of *fireController*.
- *Controllers*. Five controllers are represented on Figure 8.2. The *fireController* which receives notifications from the sensor *fireDetectorSensor* and forward them to *sprinklerController*, *TwitterController*, and *TurnstileController*. These controllers implement the

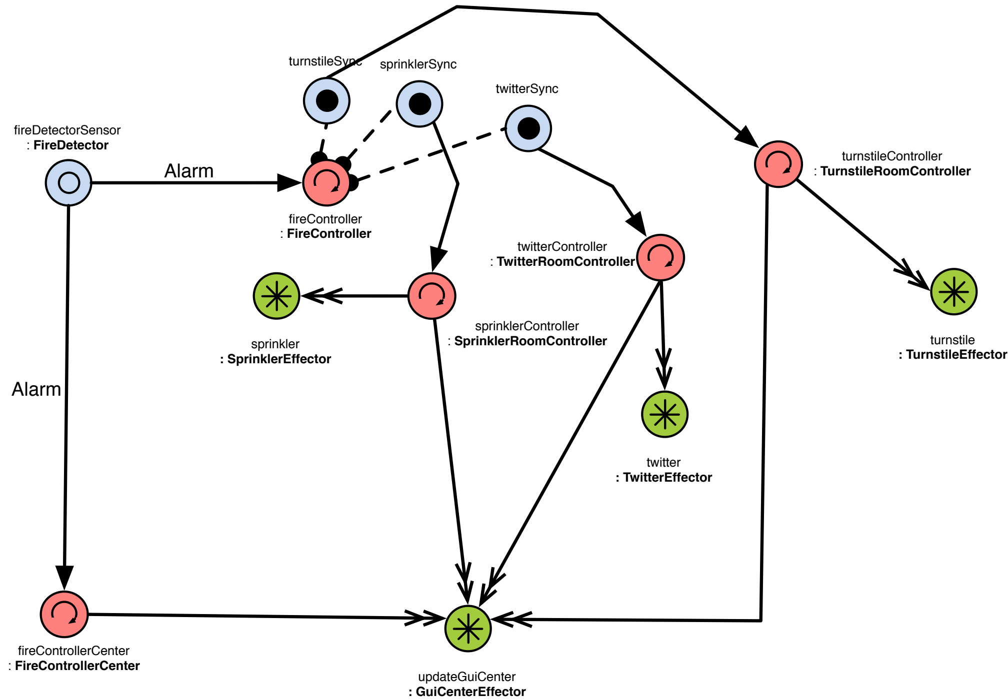

Figure 8.2: Architecture of the fire-emergency Control System

decision logic for deciding to open turnstile, trigger sprinkler or send a Twitter message. The *fireControllerCenter* also receives notifications from the *fireDetectorSensor* and updates the graphic interface of the control center.

- *Effectors*. Figure 8.2 depicts four effectors of the fire-emergency control architecture: *sprinkler*, *turnstile*, *twitter* and *updateGuiCenter*. The effector *updateGuiCenter* enables the interaction with the graphic interface of the control center.

Now, that we have described the fire-emergency control architecture using the SALTY graphical formalism, we need to implement a model of this architecture using the SALTY architectural language. The next section will present how to do that. In addition, we will provide some measures for comparing the code generated by *CORONA* with the code that is fully hand-coded by a developer in order to assess the cost-effectiveness of our approach.

8.4 Control System Implementation & Measures

In this section, we discuss the design of the fire-emergency control system architecture and its runtime implementation in SCA. We illustrate how annotation of the control system architecture can be used to drive the generation by the *CORONA* toolchain. But, before we

dig into the details of the implementation, we will first recall the development process of the *CORONA* approach defined in Chapter 4. Then, we explain for each step of the process the corresponding artifacts that are manipulated.

The development process in the *CORONA* approach consists of two main phases.

- *Design phase*. This phase of the process consists of two steps. The first step consists in designing the control architecture of the fire-emergency control system using the SALTY DSL. The second step consists in generating the corresponding Ecore model of the control system from the control architecture.
- *Runtime Phase*. From the Ecore model of the control architecture obtained at the end of the second step, *CORONA* toolchain generates the runtime implementation of the control system in SCA.

Design Phase

At this phase, developers design the control architecture using the SALTY DSL. The listing (cf. Listing 8.1) below provides an excerpt of the fire-emergency control architecture design.

```
1 object FireEmergency extends App with FCDMBuilder {
2
3   dataLinkType(name = "DL")
4   controlLinkType(name = "CL")
5
6   val Alarm = typedef {
7     dataType("Alarm") init { annotation("corona.typeref",
8    "impl" -> "fr.inria.exampledsdf.FireDetector")(_) }
9   }
10
11  sensorType(name = "FireDetector", dtype = Alarm)
12  effectorType(name = "GuiCenterEffector", operation = operation(int32))
13
14  //... definition of other control node types
15  //(sprinklerController, fireControllerCenter)
16
17   //the main global architecture
18   compositeType(name = "Main", main = true) init { implicit e =>
19
20    //sensors
21    sensor(name = "fireDetectorSensor", 'type' = 'FireDetector)
22
23    //controllers
24   controller(name = "fireControllerCenter", 'type' = 'FireControllerCenter)
25   controller( name = "fireController", 'type' = 'FireController)
26   controller( name = "sprinklerController", 'type' = 'SprinklerController)
27   controller(name = "twitterController", 'type' = 'TwitterController)
28   controller( name = "turnstileController", 'type' = 'TurnstileController)
29
30    //...definition of effectors
31
```

```
32      //databinding
33     dataBinding(name = "b14", source = sensorRef('fireController.turnstileSync),
34     target = dataLinkRef('turnstileController, 'input)) init { implicit => e
35     annotation("bind", "type"->"ws", "uri"->"http://localhost:9001/Turnstile" ,
36     "wsdl"->"http://demo.itemis/#wsdl.port(TurnstileService/TurnstilePort)")
37    }
38
39   dataBinding(name = "b16", source = sensorRef('fireDetectorSensor.twitterSync),
40   target = dataLinkRef('twitterController, 'input)) init { implicit => e
41   annotation("bind", "type"->"rest", "uri"->"/Twitter")
42  }
43
44 }}
```

Listing 8.1: Design of the Fire-emergency Control Architecture with the Salty DSL

The listing 8.1 depicts the fire-emergency control architecture design using the SALTY DSL. Lines 3–4 define datalink and controllink types. Datalink and controllink type are used for defining bindings (*databinding and controlbinding*) between nodes of the architecture. Lines 6–12 define the data type *Alarm*, and the node types sensorType and effectorType for the node *FireDetector* and *GuiCenterEffector* respectively. For the sake of simplicity, a complete definition of all the node types of the fire-emergency architecture is not provided in this listing. Lines 16–46 of the listing define the concrete architecture of the fire-emergency control system using element types defined earlier. In this concrete architecture, we define all the nodes of control system and connections between them. Lines 22-32 of the listing define controllers of the control system, and lines 37–45 define databinding between some nodes of the architecture. In particular, lines 37–40 depict a binding between the sensor *turnstileSync* and the controller *turnstileController*.

Listing 8.1 exhibits how reusability and the heterogeneity properties are carried out in the *CORONA* approach. Concerning the reusability, lines 7–8 show how implemented software pieces can be used for designing the control system. On these lines, the annotation *@corona.typeref* is used to specify the Java class *fr.inria.exampledsdf.FireDetector* that implements alarm signal. In the context of this scenario, the Java class *fr.inria.exampledsdf.FireDetector* as already been implemented, and using the annotation *@corona.typeref*, developers can enforce the reusability of this class when designing the control system.

Similarly, the heterogeneity of the *CORONA* approach is illustrated on lines 37–40, and 43–45 of Listing 8.1. These lines show how annotations can be used for describing binding based on heterogeneous implementation technologies. On lines 37–40, the annotation *@bind* is used to specify a binding of type *WebService* (*ws*), while on lines 43–45, it is used to specify a binding of type *REST*. These annotations are instrumented by the code generator for generating corresponding artifacts with respect to the required target technology. In the case of this scenario, *CORONA* generates SCA binding of type *WebService* and *REST*. Thanks to the FraSCAti runtime, these heterogeneous technologies can interoperate with each other.

	Generated Artifacts		Implemented Code
	Java	SCA	Java
Artifacts	13	13	0
LoC	84	112	65
Ratio LoC (%)	75		25

Table 8.1: Metrics of generated and Implemented Code

Now let us take a look at generated SCA artifacts that implements the fire-emergency control system.

Runtime Phase
The *CORONA* code generator uses the *ecore* model of the control system architecture obtained at the end of the design phase to generate runtime artifacts of the control system in SCA. Figure 8.3 depicts the assembly architecture of the fire-emergency control system in SCA.

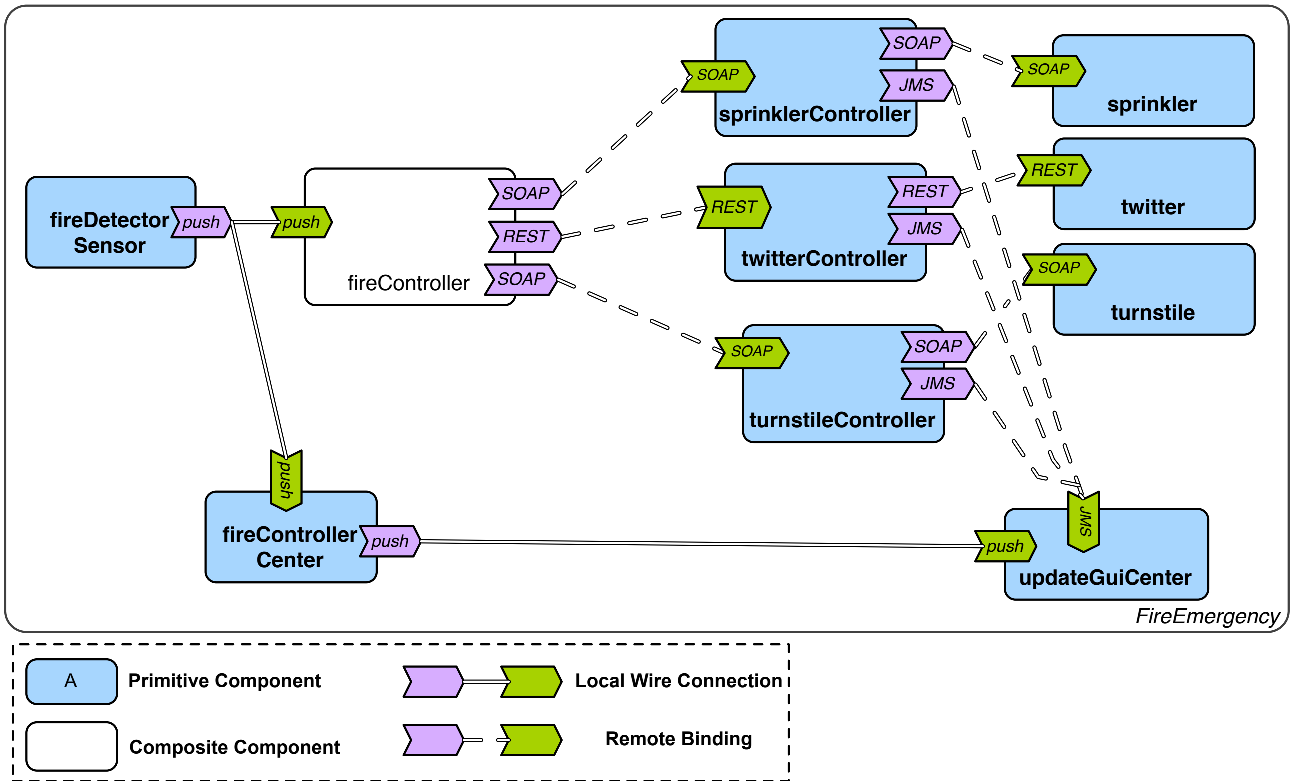

Figure 8.3: Fire-emergency SCA assembly architecture

Figure 8.3 shows that for each element of the control system depicted on Figure 8.2, *CORONA* generates a corresponding component or a composite artifact in SCA. This en-

forces the visibility of feedback control loops and the traceability between the design and the implementation of the control system. On Figure 8.3, we also observe different types of binding, and namely *SOAP, REST, JMS* bindings. Once, the implementation of the control system is generated, developers can complete this implementation to have the running instance of the control system.

Table 8.1 summarizes the metrics related to the generated and implemented code of the fire-emergency control system. Artifacts refers to classes/interfaces in Java, and composites/components in SCA. On Table 8.1, when we look at the *implemented code* column, we notice that no additional artifacts where implemented. That is because the toolchain generates all necessary artifacts required for implementing the behavior of the control system. Developers just have to complete these artifacts with the code implementing their behavior. Table 8.1 shows that for this case study developers has to write only 25% of the code implementing the control system. This is clearly a gain of time for developers of autonomic systems. This ratio can be different for other experiments, but what remains relevant to us is that efforts that are deployed for implementing autonomic systems are reduced through our approach where a portion of the required implementation code is generated.

The SCA assembly of the control architecture can be executed with the FraSCAti platform which interprets this assembly to build an instance of the fire-emergency control system. FraSCAti also provides support for dynamic inspection and reconfiguration of SCA components. Support for dynamic reconfiguration of SCA components allow to modify some architectural properties at runtime.

8.5 Summary

In this chapter we have presented the fire-emergency case-study, and used it to demonstrate the reusability and the generality of the *CORONA* approach. We have showed that by using annotations during the design of the control system architecture, developers of autonomic system can enforce the reuse of software pieces that have already been implemented. We have also showed that *CORONA* leverage the heterogeneity of various implementation technologies through annotations. We have exemplified this aspects with the *WebService* and *REST* binding in the fire-emergency control architecture. Finally, we have demonstrated that the *CORONA* approach is cost-effective because it requires less time for implementing autonomic behavior. Thanks to a generative approach, almost 75% of the code implementing the fire-emergency control system is generated.

In the next chapter, we discuss the effectiveness of using stabilization algorithms for designing control system.

Chapter 9

Smart-Mall Case-Study

"The power of accurate observation is frequently called cynicism by those who don't have it."
– George Bernard Shaw

Contents

9.1 Objective 131
9.2 Smart-Mall Scenario Description 132
9.3 Experiment & Measures 134
9.4 Summary 139

9.1 Objective

In autonomic systems, feedback control loops that control adaptive behaviors observe the environment and take decisions on the ground of these observations. For autonomic systems, the stabilization of the control system is an important issue because, on one extreme, if an autonomic manager is not reactive enough to respond to changes in the environment, it will miss some important changes that occurred in this environment. On the other extreme, if an autonomic manager is sensitive to any subtle changes that occur in the environment, this can lead the system to an unstable state. Finding the trade-off between the responsiveness and the stability is an important issue for autonomic systems.

In chapter 5, we have introduced stabilization mechanisms and a composition model for engineering control systems. We have argued that stabilization algorithms can help to build autonomic systems with a stable control system. The objective of this chapter is to assess that claim by showing how our composition model can help implementing applications with stable control systems.

Structure of the chapter
This chapter is organized as follows: In Section 9.2, we describe the *Smart-mall* scenario. Then, in Section 9.3, we present some measures that illustrate the stabilization mechanism. Finally, we conclude this chapter with a summary (cf. Section 9.4).

9.2 Smart-Mall Scenario Description

In this scenario, we consider a mall which offers several services that can be accessed from customers mobile devices. In particular, the smart-mall environment is equipped with advertising services which can collect users' preferences from their mobile devices, and provide promotional products on sale near users locations. The mall also provides other services, like a localization service that helps customers to find store locations.

In this scenario, while customers are walking through the gallery, they can discover and access services. These services are available only within their coverage area. For example, near a music store, customers can use a free multimedia service which remains accessible only around the music store area. The mall also has a *showroom*, where customers can actively try upcoming appliances.

The showroom is a smart environment equipped with smart devices like sensors (light, temperature, movement, etc.) to track changes in the environment, such as the entrance to or the exit from the showroom, and actuators for interacting with the environment. The showroom is equipped with a temperature regulation system that prevents temperature to rise above *25.5* °C. This system collects information from different sensors in the room and decides the adjustments on the air conditioner to keep the temperature under the threshold. Figure 9.1 illustrates this scenario.

From this scenario description, we consider two situations:

Situation 1
Bob is in the music store area. His mobile device executes a music player application that connects to the multimedia service and starts playing available songs according to his preferences. Because of the increasing number of connections to the store multimedia service, the quality of service starts to degrade. In particular, the service responsiveness oscillates between available and unavailable status.

Situation 2
The temperature regulation system has to consider several information to determine the temperature variation inside the showroom, and to keep it under the threshold value. For example, when appliances on test are turned on by visitors, they can generate heat or cold. In a similar way, when new people are getting in or out of the showroom, or when children stand at the doors preventing them from closing, this generates modifications of the room temperature.

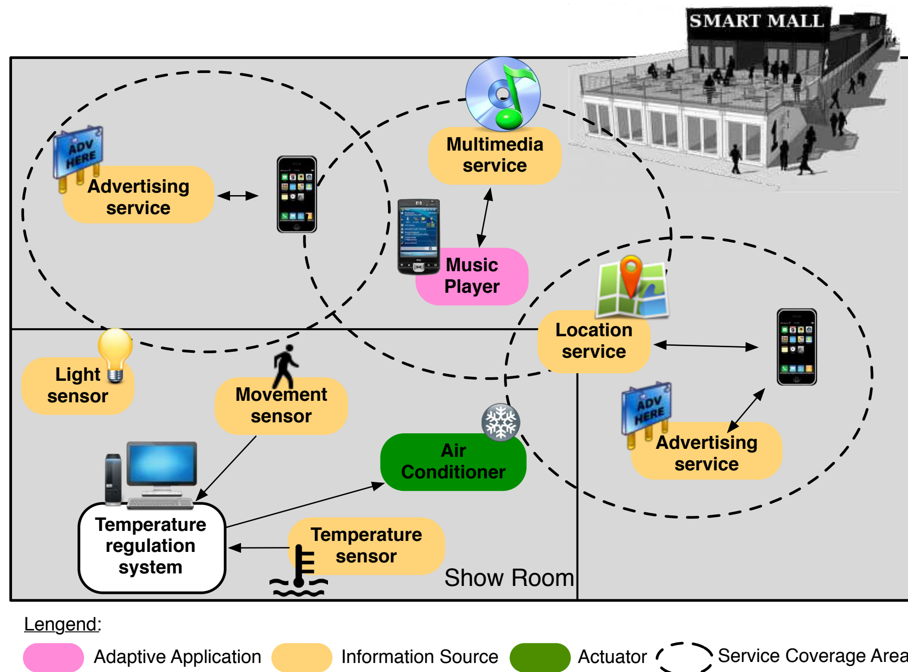

Figure 9.1: Smart-Mall Scenario illustration

Challenge

Situations described above raise some challenges concerning adaptations with regards to changes in the environment.

In the scenario, the main challenge consists in identifying relevant situations from information generated in the environment in order to adapt the behavior of the system accordingly. Hence, the relevancy of the decision-making largely depends on the quality of the collected information. For example, information *up-to-dateness*[BS03]–which refers to the time elapsed since information was produced or collected from the environment– is important for the decision-making because, as the environment evolves rapidly, decision made on old information may become useless. In the same manner, a newly updated information which does not complies with the evolution trend of the environment may induce inadequate decision-making, where an old but *accurate* information will engender smart decision-making.

The high number of connections to the store multimedia service, requires the music player application on Bob's phone to perform some adaptations to ensure that he can continue to listen to music without interruptions. For this situation, several adaptations can be envisioned. One possible adaptation can be to use a *cache*, to buffer songs downloaded from the multimedia service on Bob's phone. Another adaptation can consist in playing songs

available on the phone while the multimedia service remains overshoot. Since the availability of the multimedia service fluctuates over the time, it important for the application to adapt smartly. It would be inefficient, if the music player application triggers an adaptation each time that the multimedia service is getting available or unavailable. In particular, this can be expensive in terms of battery consumption for Bob's mobile phone.

In the section below, we discuss how the composition of stabilization algorithms can be used for selecting relevant changes in the smart-mall environment for performing adaptations.

9.3 Experiment & Measures

In order to evaluate our proposal on the composition of stabilization algorithms, we used a MacBook Pro laptop, with the following software and hardware configuration: 2.4 GHz processor, 2 GB of RAM, Mac OS X 10.5.6 (kernel Darwin 9.6.0), Java Virtual Machine 1.6.0. The experimental test bed is completely modeled with the SIAFU [MN06] simulation engine. This engine allows us to create repeatable environment for the experiment.

In this section, we present the result of two experiments corresponding to the *situation 1* and *2* of the smart-mall scenario.

A- Temperature variation in the showroom
In this experiment, we consider the monitoring of temperature variation in the showroom of the smart-mall presented in Section 9.2. This experiment explores the efficiency of composition strategies in an environment characterized with important variations of the observable context. In the scenario, many factors contribute to temperature variation inside the showroom: doors that open and close as people getting in and out, appliances activities that generate heat or the number of people gathered in the room. To keep the temperature below the target threshold ($25.5°C$), the temperature regulation system of the showroom has to adapt its behavior according to these changes. In particular, the regulator system should be able to detect when temperature is within the critical range of $24°C$ to $25.5°C$ in order to perform required actions (*e.g*, to increase heat or decrease cold). We assume that data retrieved from temperature sensors has an uncertainty of $0.1°C$. The frequency for collecting data from sensors is $300ms$.

Figure 9.2 illustrates the control system architecture for regulating temperature variation in the showroom. The control system is composed of three sensors (*lightDetectorSensor, movementDetectorSensor, temperatureDetectorSensor*), one controller (*temperatureController*), and one effector (*conditionerEffector*). The annotation @stabilized for the connection between the *temperatureDetectorSensor* and the *temperatureController* indicates the use of a stabilization algorithm *DeltaOperator*. The latter intends to stabilize the variation of input values from the temperature sensor.

The experiment is realized in three steps. First, we run the experiment with the *Delta Operator* (DO) [BSBF02] stabilization algorithm, which is associated to high responsiveness

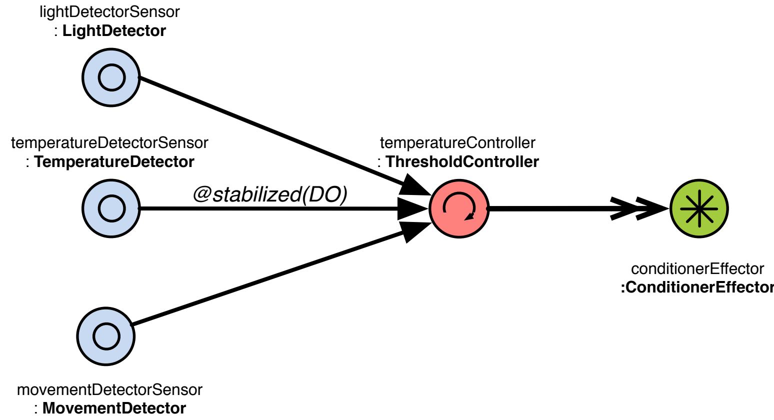

Figure 9.2: Showroom Control System Architecture

according to our classification (cf. Chapter 5). Second, we rerun the experiment with the *Kalman Filter* (KF) algorithm, which refers to the class of less responsive algorithms. And finally, we run the experiment by applying an horizontal composition strategy for DO and KF algorithms. We used the following composition rule (CR) : $Max\,(Min\,(T_{do_{t-1}}, T_{do_t}),\, T_{kf_t})$, where $T_{do_{t-1}}, T_{do_t}$ are temperature values from DO at time $t-1$ and t of the execution, and T_{kf_t} is the temperature value from KF at time t of the execution. The transition matrix A for Kalman Filter algorithm was defined as follows: given x_{k-1}, z_{k-1}, R, the prediction, the value of the measured variable at $k-1$ step, and the variance respectively.

We define ε, Δ and A as follows: $\Delta = x_{k-1} - z_{k-1}, \varepsilon = R * z_{k-1}$

$$A_K = \begin{cases} A_{k-1} & for \quad |\Delta| \leq \varepsilon \\ \frac{z_k - \varepsilon}{x_{k-1}} & for \quad |\Delta| > \varepsilon \quad and \quad \Delta < 0 \\ \frac{z_k + \varepsilon}{x_{k-1}} & for \quad |\Delta| > \varepsilon \quad and \quad \Delta > 0 \end{cases}$$

With the assumption that temperature in the showroom changes rapidly, it is challenging to choose the right stabilization mechanism in order to filter out *false positive* events. A *false positive* event can be observed for example, when for two consecutive readings of temperature sensors, the first value is within the critical range and the second value is out of it. In general, a *false positive* event can be considered as an isolated event that does not reflect the global trend of the system behavior.

The results of this experiment are reported in Figures 9.3 and 9.4. Figure 9.3 depicts 3 curves representing the application behavior without stabilization, stabilized with KF, and stabilized with DO. The two horizontal lines on the figure indicate the critical range. Each point in the critical area corresponds to a triggered adaptation. The behavior of KF and DO on the Figure is conformed to our classification, which associate DO with a high responsiveness and KF with a low responsiveness. This is confirmed by the curve of the application stabilized with KF. That curve shows few points within the critical range, which means that

few adaptations were triggered. On contrary, the curve of the application stabilized with DO has more points than the former.

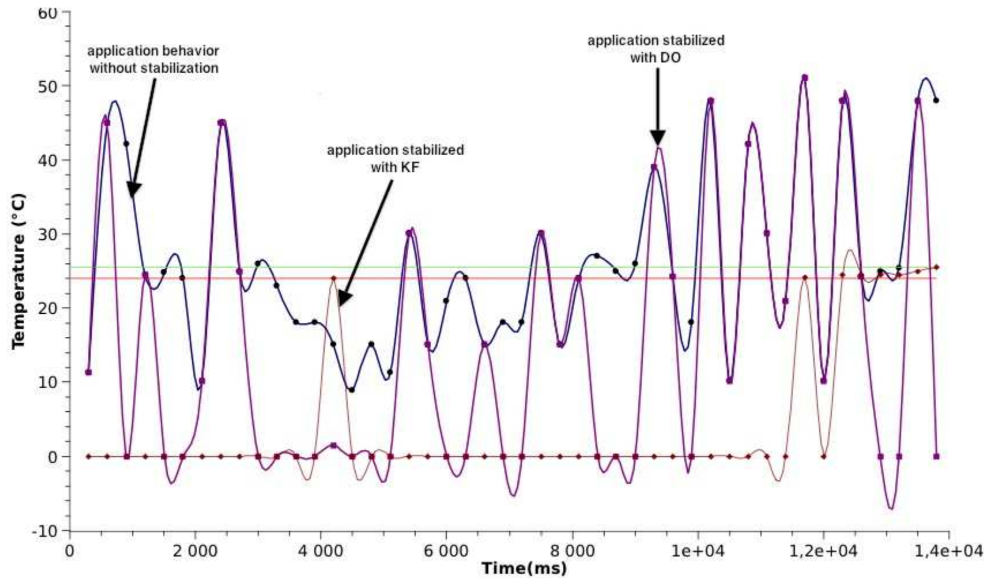

Figure 9.3: Stabilized application behavior with Kalman Filter or Delta Operator

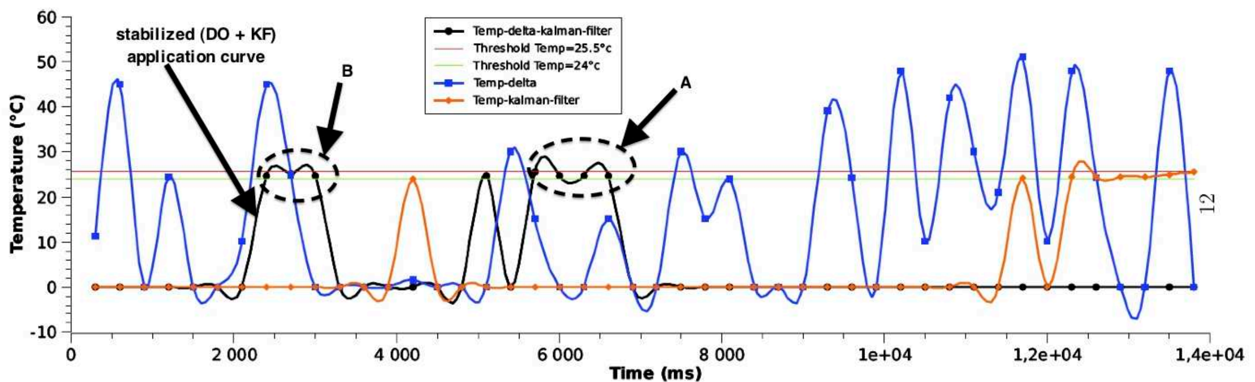

Figure 9.4: Stabilized application behavior with Horizontal Composition (DO+KF)

In the third step of this experiment, we evaluate the behavior of the application stabilized with a single stabilization algorithm (DO or KF), and with both algorithms using *horizontal composition* strategy. Figure 9.4 compares curves of the stabilized application behavior. The Arrows on the Figure indicate the curve of the application stabilized using the *horizontal composition* strategy. The others curves correspond to the application behavior stabilized with DO and KF as presented in Figure 9.3. On Figure 9.4, two zones of the curve

(DO+KF) are represented by an ellipse. On that Figure, we notice that the curve (DO+KF) has its first event within the critical area (cf. zone B) earlier ($t = 2.5s$) than the curve (KF), which has it first event at time $t = 4.2s$. This can be explained by the fact that, by combining both algorithms, the resulting application benefits from the responsiveness of DO and becomes more responsive than KF algorithm alone. We observe that, for the curve (DO+KF), events in the adaptation area are grouped. This can also be explained by the fact that, by combining both algorithms, the resulting application benefits from KF's prediction property which causes the system to moderate variations of the input data. These observations tend to demonstrate that the composition of stabilization mechanisms offers a good compromise in terms of responsiveness for pervasive applications in fast changing context.

The horizontal composition of stabilization algorithms gives the best result in this scenario situation. By combining algorithms with different responsiveness (DO, KF) we were be able to detect relevant temperature variations within the critical area.

B- Media service availability near music store

This experiment focuses on the adaptation of Bob's media player application around the music store of the smart-mall scenario. In the scenario, because of the high number of connections, the store multimedia service is sometime unavailable. Bob's music player needs to adapt according to the volatility of the store multimedia service. In addition, these adaptations have some impacts on Bob's phone battery. Therefore, it is important for the player to perform smart adaptations in order to ensure that Bob can listen to music with few interruptions, and that phone's battery last. The purpose of this experiment is to demonstrate how using vertical composition strategy helps at performing smart adaptations in a volatile context.

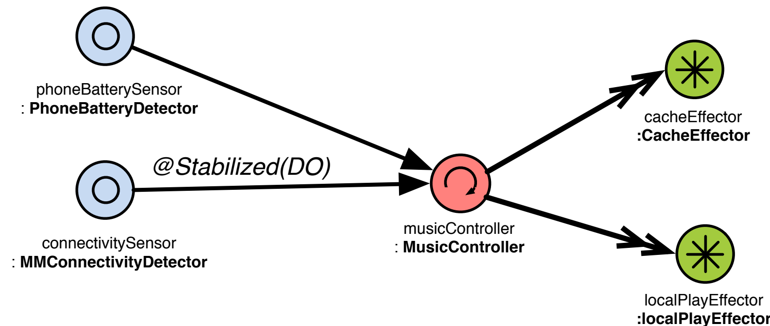

Figure 9.5: Media Player Control System Architecture

Figure 9.5 illustrates the control system architecture of bob's phone multimedia player. The control system is composed of two sensors (*connectivitySensor, phoneBatterySensor*), one controller (*musicController*), and two effectors (*localPlayEffector, cacheEffector*). The annotation @stabilized for the connection between the *connectivitySensor* and the *musicController* indicates the use of a stabilization algorithm *DeltaOperator*. The latter intends to stabilize the variation of input values from the sensor connectivitySensor. The effector *localPlayEffector*

enforces the player to play music from the phone's media library, while the effector *cacheEffector* activate the use of a buffer for caching information downloaded from the multimedia service.

To assess the behavior of stabilizations algorithms in a volatile context, we randomly generate bandwidth values to characterize variations of the media service availability. Then, we evaluate the number of adaptations triggered by the system for three configuration cases: *(i)* Application running without any stabilizations mechanisms, *(ii)* Application running with DO algorithm and, *(iii)* Application running with vertical composition strategy of DO and *SimpleFuzzy*.

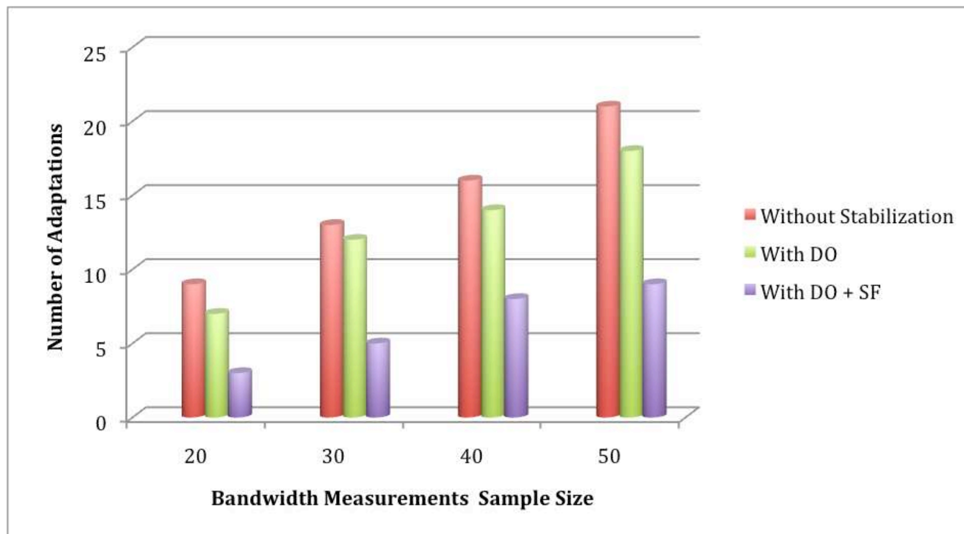

Figure 9.6: Variation of Triggered Adaptations

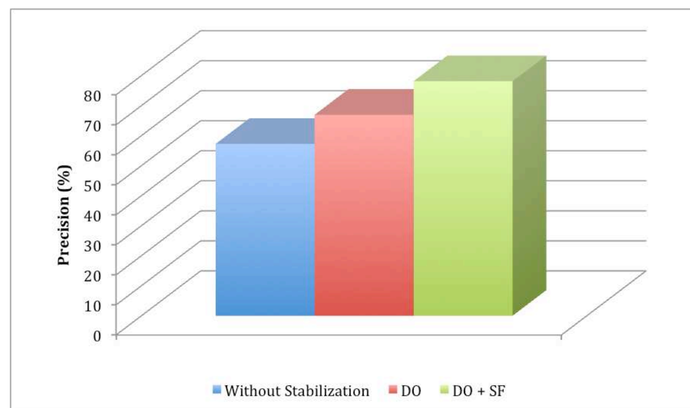

Figure 9.7: Precision of Context Stabilization

Figure 9.6 gives the variations of the number of adaptations triggered by the applications for three configuration cases. The experiment is repeated for various samples of bandwidth measurements. This figure shows that the lowest rate of adaptations are triggered for the case when the application is working with the composition of DO and SF algorithms. That is because, the application working with DO and SF is less responsive to the variation of the bandwidth. When the application's goal is to minimize Bob's phone battery usage, the configuration case (iii) should be the most suitable in order to achieve it. That is because, the more the application needs to adapt the more it consumes battery. However, having few adaptations triggered by the application (case iii), does not reveal anything about the quality of these adaptations. This raises the following questions: does the application was able to detect relevant adaptation situations? What is the percentage of false positive adaptations detected by the application?

To answer the above questions, we completed this experiment and realized additional measurements on the precision and recall of the context stabilization. For that purpose, we injected adaptation situations in the measure samples, and try to evaluate how many of them the application was able to retrieve. *Precision* P_{r_i} and *Recall* R_i are calculated as a percentage of the following formulas:

$$P_{r_i} = \frac{|\{Injected\,adaptation\;situations\} \cap \{Retrieved\,adaptations\}|}{|\{Retrieved\,adaptations\}|}$$

$$R_i = \frac{|\{Injected\,adaptation\;situations\} \cap \{Retrieved\,adaptations\}|}{|\{Injected\,adaptation\;Situations\}|}$$

Figure 9.7 summarizes the result of the average precision for the context stabilization. The recall calculated is 100% for all the configurations. It means that, all injected adaptation situations were successfully retrieved during the experiment. However, as depicted on Figure 9.7, the highest precision is reached for the configuration case (iii). From that we can infer that the vertical composition of DO + SF increases the accuracy of decisions for adapting.

One can argue that similar results can be obtained without our composition model. Indeed, the idea of combining stabilization algorithms is not new [SAH07]. We can easily find applications that rely on the combination of several stabilization mechanisms. However, in all these applications the implementation of stabilization mechanisms remains static, and ties to the application architecture. In our approach the stabilization algorithms participating in the decision making can be adapted. This means that, they can be changed, and the way they are interacting as well.

9.4 Summary

In this chapter, we have presented the smart-mall scenario experiment to validate the importance of stabilization algorithms in order to implement stable control systems. In particular,

we have evaluated the efficiency of our stabilization algorithms composition model. We have showed that the *horizontal composition* of stabilization algorithms with various responsiveness class enable to detect relevant changes of the context in a fast changing environment. In the same way, we have demonstrated that a *vertical composition* of stabilization algorithm increases the accuracy of decisions for adapting applications with regard to changes in the environment.

In the next chapter, we start the final part of this thesis, and present the conclusion and the perspectives of this dissertation.

Part IV

Conclusion & Persperctives

Chapter 10

Conclusion

"However long the night, the dawn will break." unknown author

Contents

10.1 Summary of the Dissertation **143**

10.1.1 Objectives vs. Contributions 145

10.2 Perspectives **146**

10.2.1 Short Term Perspectives 146

10.2.2 Long Term Perspectives 147

This chapter summarizes the main contributions of this dissertation and highlights some research perspectives of this work. The chapter is organized as follows: We first give a summary of this dissertation (cf. Section 10.1), then we conclude the chapter by discussing some perspectives (Section 10.2).

10.1 Summary of the Dissertation

The increasing complexity of modern software systems has pushed the research community to investigate innovative paradigms for engineering, managing, and deploying them. Autonomic computing which promotes the vision of software systems that are able to manage themselves in an autonomous manner seems to offer an approach to address this complexity in an effective way. As a consequence, this last decade, autonomic computing has become a major research topic that has mobilized both the research and the industry communities.

In addition to the software complexity, modern software systems must satisfy requirements in term of software qualities. For example, the growing importance for environmental and sustainable development concerns at the socio-political level has created the need for

energy-efficient software. Therefore, the challenge that stands ahead autonomic researchers is to provide software engineers with techniques for engineering amenable autonomic software systems that takes into account energy effeciency.

In the state-of-the-art (cf. Part I), we have seen that many autonomic solutions were proposed by the industry and the research community. But successful solutions are scarce and partially satisfy engineering requirements for building amenable autonomic systems. In particular, concerning architecture-based autonomic solutions which have been the scope of interest during this thesis, feedback control loops that implement control systems merely serve as blueprint for the design of autonomic solutions. In case they are explicitly used for designing control systems, the runtime implementation of the control system does not reflect the design. This disruption between the design and the runtime implementation of control systems as been reported in specialized conferences and papers [ST09, MPS08, CdLG+, BSCG+09] as one of the major limitation for building amenable autonomic systems. One of the main reasons that justifies the importance of visible control systems architecture is the need for building dependable autonomic systems. One solution for filling this need is to implement verification techniques for validating autonomic systems. However, the lack of visibility is the main limitation for implementing such techniques.

Nevertheless, the visibility of feedback control loops is not enough, it is just a mean towards building amenable autonomic systems. Throughout this dissertation, we have addressed the visibility challenge and propose a new approach, *CORONA*, for engineering amenable autonomic systems. In the *CORONA* approach, the challenge of building autonomic systems with visible feedback control loops is achieved by relying on the SALTY model for designing control systems. The SALTY model provides a control-oriented syntax where feedback control loops are represented as first-class concerns. In addition, to enhance the traceability between the control system design and its runtime implementation, we have developed the *CORONA* toolchain that ensures a strong mapping between the design and the implementation of the control system. In the *CORONA* approach, to guide the implementation of the control system, the *CORONA* toolchain generates a part of the implementation code of the control system which conforms to the design. This accelerates the development process of autonomic solutions, and reduces the overall cost of implementing autonomic solutions.

Furthermore, in *CORONA* we have identified and modeled conflicts patterns that occur when implementing control systems. We have implemented verification algorithms for automated checking of the control system architecture model against these conflicts, and developed some algorithms for the automatic resolution of some of them. As a consequence, our approach enables developers of autonomic systems to build more reliable software solutions.

Finally, we have shown how non-functional properties can be seamlessly integrated during the design of the control system architecture in order to deliver required quality attributes. In particular, we have focused our research on the engineering of autonomic systems with a *stable* control. For that purpose, we have proposed a composition model of sta-

bilization algorithms based on the qualitative characterization of the later. Our composition model consists mainly in two strategies: the *horizontal* composition, and the *vertical* composition. Horizontal composition enables to increase the responsiveness of the control system without going unstable. Vertical composition enables to increase the accuracy of that responsiveness. Stabilization mechanisms empower developers to build autonomic solutions with a *stable* control system.

10.1.1 Objectives vs. Contributions

In this section we discuss the contributions with regards to the objectives defined at the beginning of this thesis.

In chapter 4 of this thesis, we have defined the following objectives for this work:

- **Domain agnosticism.** With this objective, we aimed at defining a *generic* solution for engineering autonomic systems. In our proposal, the design of the control system is based on the SALTY model which provides a platform-agnostic syntax. As a consequence, a control system architecture design can be used for different implementation platform. In the fire-emergency case-study (cf. Chapter 8), we have illustrated with the REST and SOAP (WebService) binding, how different technologies can been reached from the unique description of the control architecture. Agnosticism with regards to the implementation platforms enhances the reusability of the control system architecture.

- **Transparency.** This objective aims at providing support for reasoning about self-adaptive capabilities of autonomic systems at design and at runtime. That is, autonomic systems must explicitly describe their control flow. Transparency is particularly required in the context of multiple control systems with multiple objectives. In *CORONA* this objective is achieved in two steps: First, by providing a syntax for designing control systems where feedback control loops are reified as first-class concerns. Secondly, by ensuring a strong mapping between the design and the implementation of the control system.

- **Cost-effectiveness**. This objective aims at reducing the cost for engineering and maintaining autonomic systems. This objective is achieved in our proposal in three different ways: First, the *visibility* of the feedback control loops that govern autonomic systems, increases their maintainability, because the objective of each control system can be easily identified. Secondly, the use of *generative techniques* for generating part of the control system implementation code, reduce the burden of implementing autonomic solutions for developers. Finally, the support for *conflicts-free* control systems through automatic checking of the control architecture model enforces the implementation of more reliable autonomic solutions.

- **Modularity.** This objective aims at building *flexible* and *scalable* autonomic solutions. In *CORONA* we have addressed this objective by promoting a decentralized control architecture approach. That is, the control system is build from the composition of feedback control loops. From there, various style of collaboration can be envisioned for building the control system architecture. The second level for supporting modularity in *CORONA* is the transparent integration of cross cutting-concerns in the control system architecture. This is behavior is mainly implemented by enriching the control system architecture model with specialized annotations construction.

10.2 Perspectives

In this dissertation, we have presented our vision and our solution towards building amenable autonomic systems. However, there are still some aspects where our work needs to be completed. In this section, we present the short-term and the long-term perspectives that appears to be the must relevant for the continuation of this work.

10.2.1 Short Term Perspectives

Conflicts patterns modelisation

In this thesis, we have identified some conflicts patterns that occur when designing control systems. We think that the identification of new conflicts patterns is important for building more reliable autonomic systems. In particular, the identification of other types of *indirect overlaps* (cf. Chapter 6 Section 6.4.2) patterns seems to be a promising research direction. We also think that the identification of new kind of conflict patterns will raise some issues related to the domain of analysis for interpreting the meaning of analysis results. Finally, we believe that it is relevant to work on automated reconciliation strategies to resolve conflicts detected in the control system architecture.

Quality attributes for verification tools analysis results

Another aspect that can be interesting to investigate as a continuation of this work relates with quality attributes for verification tools. This perspective fits in the vision of increasing the understandability of results provided by verification tools. Indeed, in order to leverage the result of verification tools, we must investigate the option of providing a way to qualify and categorize these verifications. For example, by indicating the coverage rate of the analysis like we do for software testing in common programming languages.

Automatic test generation

Automatic test generation for evaluating the execution behavior of the control system in autonomic systems is another direction to investigate towards building amenable autonomic system. During this dissertation, we have observed that there is a lack of tools and established methodologies for testing the behavior of control systems. This is partly due to the heterogeneity of managed systems which can be costly to reproduce on physical machines.

We think that, the use of simulation tools can offer some first answer to that issue. During this dissertation, we have demonstrated the uses of some simulation tools for testing the behavior of control systems. The next step in that perspective consists in automating the generation of simulation testbeds from the architecture design of the control system.

Graphical user interface
Thanks to the SALTY model, developers of autonomic systems can easily design control systems with a syntax that reifies feedback control loops as first-class concerns. However, we think that a graphical user interface that can enable designers of control systems to draw an architecture and obtain an automatic generation of the design code in the SALTY language can significantly enhance the work of the later. In addition, this will improve the engineering process of implementing autonomic systems through the *CORONA* approach.

10.2.2 Long Term Perspectives

Implementing -ASO[5] properties of autonomic systems
One of the key contribution of this dissertation is the attempt of bringing into software engineering some good principles of the control theory for engineering feedback control loops. In particular, we have demonstrated how the use of stabilization mechanisms can helps implementing autonomic systems with a stable control. However, beside the stability, the *accuracy*, the *settling time* and the *overshoot* are others properties used by control engineers to calibrate the behavior of feedback control loops. Similarly to what we have done for the *stability*, one perspective of this dissertation can be to investigate to which extends *accuracy*, *settling time*, and *overshoot* can be used in software engineering for implementing control systems.

Consistency for dynamic reconfiguration of the control system architecture
Thanks to the FraSCAti runtime, control systems implemented with the *CORONA* approach can be dynamically adapted and reconfigured. The dynamic adaptation of the control system architecture can introduce some inconsistencies that can generate side-effects behaviors. As a consequence, it is important to provide support for runtime checking of the control system in order to ensure that it remains conforms to design constraints after an adaptation. We believe that the consistency of dynamic and adaptable control systems is a relevant perspective that derives from this dissertation.

[5] Refers to the SASO (stability, accuracy, settling time, overshoot) properties in control engineering presented in Chapter 5

This page was intentionally left blank

Bibliography

[AAG93] Gregory Abowd, Robert Allen, and David Garlan. Using style to understand descriptions of software architecture. *SIGSOFT Softw. Eng. Notes*, 18(5):9–20, December 1993. 16

[APJ+03] Michael Atighetchi, Partha P. Pal, Christopher C. Jones, Paul Rubel, Richard E. Schantz, Joseph P. Loyall, and John A. Zinky. Building Auto-Adaptive Distributed Applications: The QuO-APOD Experience. In *Distributed Computing Systems Workshops*, pages 104–109. IEEE, May 2003. 74

[Apt03a] Krzysztof Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003. 48

[Apt03b] Krzysztof Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003. 84

[ASHP+08] Ahmad Al-Shishtawy, Joel Höglund, Konstantin Popov, Nikos Parlavantzas, Vladimir Vlassov, and Per Brand. Distributed control loop patterns for managing distributed applications. In *Proceedings of the 2008 Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops*, SASOW '08, pages 260–265, Washington, DC, USA, 2008. IEEE Computer Society. 14

[Bar04] Bart Jacob, Richard Lanyon-Hogg, Devaprasad K Nadgir, Amr F Yassin. *A Practical Guide to the IBM Autonomic Computing Toolkit*. IBM Redbooks, Apr 2004. 18

[BCS02] E. Bruneton, T. Coupaye, and J. B. Stefani. Recursive and dynamic software composition with sharing. 2002. 16

[BG01] Jean Bézivin and Olivier Gerbé. Towards a precise definition of the omg/mda framework. In *Proceedings of the 16th IEEE international conference on Automated software engineering*, ASE '01, pages 273–, Washington, DC, USA, 2001. IEEE Computer Society. 24

[BHRE07] Gunnar Brataas, Svein Hallsteinsen, Romain Rouvoy, and Frank Eliassen. Scalability of Decision Models for Dynamic Product Lines. In *Proceedings of the International Workshop on Dynamic Software Product Line (DSPL)*, pages 23–32, September 2007. 74

[BHS+08] Laurent Broto, Daniel Hagimont, Patricia Stolf, Noel Depalma, and Suzy Temate. Autonomic management policy specification in tune. In *Proceedings of the 2008 ACM symposium on Applied computing*, SAC '08, pages 1658–1663, New York, NY, USA, 2008. ACM. 18, 29

[BM02] Rajkumar Buyya and Manzur Murshed. Gridsim: A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing. *concurrency and computation: practice and experience (CCPE)*, 14(13):1175–1220, 2002. 116

[BPHT06] Sara Bouchenak, Noel De Palma, Daniel Hagimont, and Christophe Taton. Autonomic management of clustered applications. In *CLUSTER*, 2006. xi, 16, 17, 29

[BS03] Thomas Buchholz and Michael Schiffers. Quality of context: What it is and why we need it. In *In Proceedings of the 10th Workshop of the OpenView University Association: OVUAÓ3*, 2003. 68, 133

[BSBF02] Céline Boutros Saab, Xavier Bonnaire, and Bertil Folliot. PHOENIX: A Self Adaptable Monitoring Platform for Cluster Management. *Cluster Computing*, 5(1):75–85, 2002. 74, 78, 134

[BSCG+09] Yuriy Brun, Giovanna Di Marzo Serugendo, Holger Giese Cristina Gacek, Holger Kienle, Hausi Müller Marin Litoiu, Mauro Pezzè, and Mary Shaw. Engineering Self-Adaptive Systems through Feedback Loops. *Software Engineering for Self-Adaptive Systems (SELfSAS)*, 5525:48–70, 2009. 30, 46, 144

[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(3):350–371, July 2002. 18

[CBCL11] Damien Cassou, Emilie Balland, Charles Consel, and Julia Lawall. Leveraging software architectures to guide and verify the development of sense/compute/control applications. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 431–440, New York, NY, USA, 2011. ACM. 24, 29

[CdLG+] Betty Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Giovanna Di Marzo Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, Holger Kienle, Jeff

Kramer, Marin Litoiu, Sam Malek, Raffaela Mirandola, Hausi Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns, and Jon Whittle. Software engineering for self-adaptive systems: A research roadmap. In Betty Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee, editors, *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science*, pages 1–26. Springer Berlin/Heidelberg. 30, 34, 46, 144

[CSW04] David M. Chess, Alla Segal, and Ian Whalley. Unity: Experiences with a prototype autonomic computing system. In *Proceedings of the First International Conference on Autonomic Computing*, ICAC '04, pages 140–147, Washington, DC, USA, 2004. IEEE Computer Society. 18, 29

[Dar07] Waltenegus Dargie. The Role of Probabilistic Schemes in Multisensor Context-Awareness. In *5th IEEE International Conference on Pervasive Computing and Communication workshops (PerCom'07)*, pages 27–32. IEEE, 2007. 72, 74

[DB00] Richard C. Dorf and Robert H. Bishop. *Modern Control Systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 9th edition, 2000. 12

[dro] DROOLS Rule Engine. `http://www.jboss.org/drools/`. 78

[dRRM06] Couto Antunes da Rocha, Ricardo, and Endler Markus. Middleware: Context Management in Heterogeneous, Evolving Ubiquitous Environments. *IEEE Distributed Systems Online*, 7(4), 2006. 72, 74

[Fab] Fabric. Python library and command-line tool. `http://docs.fabfile.org/en/1.4.3/index.html`. 49

[FDDM08] Areski Flissi, Jérémy Dubus, Nicolas Dolet, and Philippe Merle. Deploying on the Grid with DeployWare. In *Proceedings of the 8th International Symposium on Cluster Computing and the Grid (CCGRID)*, pages 177–184. IEEE, May 2008. 49

[GS02] David Garlan and Bradley Schmerl. Model-based adaptation for self-healing systems. In *Proceedings of the first workshop on Self-healing systems*, WOSS '02, pages 27–32, New York, NY, USA, 2002. ACM. 34

[HDPT04] Joseph L. Hellerstein, Yixin Diao, Sujay Parekh, and Dawn M. Tilbury. *Feedback Control of Computing Systems*. John Wiley & Sons, 2004. 70, 72

[HGB10] Regina Hebig, Holger Giese, and Basil Becker. Making control loops explicit when architecting self-adaptive systems. In *Proceedings of the second international workshop on Self-organizing architectures*, SOAR '10, pages 21–28, New York, NY, USA, 2010. 94

[HS06] Michael G. Hinchey and Roy Sterritt. Self-managing software. *IEEE Computer*, 39(2):107–109, 2006. 16

[IBM01] IBM. *Autonomic Computing: IBM's Perspective on the State of Information Technology*. IBM Research Center, 2001. 1, 13

[JCL11] Henner Jakob, Charles Consel, and Nicolas Loriant. Architecturing conflict handling of pervasive computing resources. In *DAIS*, pages 92–105, 2011. 25

[jes] JESS Java Engine Rule. `http://www.jessrules.com/`. 78

[KBE99] M.M. Kokar, K. Baclawski, and Y.A. Eracar. Control theory-based foundations of self-controlling software. *Intelligent Systems and their Applications, IEEE*, 14(3):37–45, 1999. 55

[KC03] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, January 2003. 15

[KCH+90] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, ESD-90-TR-222, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, November 1990. 76

[Ken02] S. Kent. Model driven engineering. In *Integrated Formal Methods*, pages 286–298. Springer, 2002. 33

[Kep05] Jeffrey O. Kephart. Research challenges of autonomic computing. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 15–22, New York, NY, USA, 2005. ACM. xi, 12, 13, 14

[KM05] Gregor Kiczales and Mira Mezini. Separation of concerns with procedures, annotations, advice and pointcuts. In Andrew Black, editor, *ECOOP 2005 - Object-Oriented Programming*, volume 3586 of *Lecture Notes in Computer Science*, pages 733–733. Springer Berlin / Heidelberg, 2005. 67

[KS10] Krzysztof Kuchcinski and Radoslaw Szymanek. Jacop - java constraint programming solver, 2010. 89

[LPH04] H. Liu, M. Parashar, and S. Hariri. A component-based programming model for autonomic applications. In *Autonomic Computing, 2004. Proceedings. International Conference on*, pages 10–17. IEEE, 2004. 46

[MDL10] Yoann Maurel, Ada Diaconescu, and Philippe Lalanda. Ceylon: A service-oriented framework for building autonomic managers. In *Proceedings of the 2010 Seventh IEEE International Conference and Workshops on Engineering of Autonomic and Autonomous Systems*, EASE '10, pages 3–11, Washington, DC, USA, 2010. IEEE Computer Society. xi, 22, 23, 29

[MH04] Julie A. McCann and Markus C. Huebscher. Evaluation issues in autonomic computing. In *GCC Workshops*, pages 597–608, 2004. 26

[MN06] Miquel Miquel and Petteri Nurmi. A Generic Large Scale Simulator for Ubiquitous Computing. In *3rd Annual International Conference on Mobile and Ubiquitous Systems*, pages 1–3. IEEE, 2006. 134

[MPS08] Hausi Müller, Mauro Pezzè, and Mary Shaw. Visibility of Control in Adaptive Systems. In *Proceedings of the 2nd international workshop on Ultra-Large-Scale Software-Intensive Systems (ULSSIS)*, pages 23–26, 2008. 46, 67, 144

[MRF03] Rene Mayrhofer, Harald Radi, and Alois Ferscha. Recognizing and Predicting Context by Learning from User Behavior. In *International Conference on Advances in Mobile Multimedia*, pages 25–35, September 2003. 73

[Mur04] R. Murch. *Autonomic computing*. IBM Press, 2004. 13

[NRS09] Russel Nzekwa, Romain Rouvoy, and Lionel Seinturier. Towards a Stable Decision-Making Middleware for Very-Large-Scale Self-Adaptive Systems. In *BElgian-NEtherlands software eVOLution seminar (BENEVOL)*, Louvain-la-Neuve, Belgium, 2009. 64

[NRS10] Russel Nzekwa, Romain Rouvoy, and Lionel Seinturier. A Flexible Context Stabilization Approach for Self-Adaptive Applications. In *Proceedings of the 7th IEEE Workshop on Context Modeling and Reasoning (CoMoRea)*, page 6, March 2010. 120

[OAS07] OASIS Open CSA. Service Component Architecture (SCA), March 2007. http://www.oasis-opencsa.org/sca. 49, 56, 57

[ORA] ORACLE. Java Management Extensions (JMX). http://www.oracle.com/technetwork/java. 21

[pl] Scala programming language. http://www.scala-lang.org/. 39

[PZLB05] Amir Padovitz, Arkady Zaslavsky, Seng Wai Loke, and Bernard Burg. Maintaining Continuous Dependability in Sensor-Based Context-Aware Pervasive Computing Systems. In *38th Annual Hawaii International Conference on System Sciences (HICSS'05)*. IEEE Computer Society, 2005. 73, 74

[SAH07] Odysseas Sekkas, Christos B. Anagnostopoulos, and Stathes Hadjiefthymiades. Context Fusion Through Imprecise Reasoning. In *IEEE International Conference on Pervasive Services*, pages 88–91. IEEE, 2007. 65, 72, 73, 74, 139

[SH05] Roy Sterritt and Mike Hinchey. Autonomic computing " panacea or poppycock? In *Proceedings of the 12th IEEE International Conference and Workshops on Engineering of Computer-Based Systems*, ECBS '05, pages 535–539, Washington, DC, USA, 2005. IEEE Computer Society. 13

[SMF+09] Lionel Seinturier, Philippe Merle, Damien Fournier, Nicolas Dolet, Valerio Schiavoni, and Jean-Bernard Stefani. Reconfigurable SCA Applications with the FraSCAti Platform. In *6th IEEE International Conference on Service Computing (SCC)*, pages 268–275. IEEE, 2009. 50, 56

[SMR+12] Lionel Seinturier, Philippe Merle, Romain Rouvoy, Daniel Romero, Valerio Schiavoni, and Jean-Bernard Stefani. A Component-Based Middleware Platform for Reconfigurable Service-Oriented Architectures. *Software: Practice and Experience*, 42(5):559–583, May 2012. 57

[ST09] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4(2):14:1–14:42, May 2009. 46, 144

[TC04] Lynda Temal and Denis Conan. Failure, Connectivity and Disconnection Detectors. In *1st French-speaking Conference on Mobility and Ubiquity Computing (UbiMob'04)*, pages 90–97. ACM, 2004. 72, 74

[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. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 1*, AAMAS '04, pages 464–471, Washington, DC, USA, 2004. IEEE Computer Society. 18, 29

[TTL05] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the condor experience: Research articles. *Concurr. Comput. : Pract. Exper.*, 17(2-4):323–356, February 2005. xii, 85, 86

[wC08] Shang wen Cheng. *Rainbow: Cost-effective Software Architecture-based Self-adaptation*. PhD thesis, Carnergie Mellon University, 2008. xi, 21, 22, 29

[wCcHG+04] Shang wen Cheng, An cheng Huang, David Garlan, Bradley Schmerl, and Peter Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *IEEE Computer*, 37:46–54, 2004. 21, 29

[Wu03] Huadong Wu. *Sensor Data Fusion for Context-aware Computing using Dempster-shafer Theory*. PhD thesis, Carnergie Mellon University, Pittsburgh, 2003. 77

[XSL+03] Dong X., Hariri S., Xue L., Chen H., Zhang M., Pavuluri S., and Rao S. Autonomia: an autonomic computing environment. In *Proceedings of the Performance, Computing, and Communications IEEE International Conference 2003*, pages 61–68, 2003. 20, 29

[YCHP05] Jingmei Yang, Huoping Chen, Salim Hariri, and Manish Parashar. Autonomic runtime manager for adaptive distributed applications. In *Proceedings of the High Performance Distributed Computing, 2005. HPDC-14. Proceedings. 14th IEEE*

International Symposium, pages 69–78, Washington, DC, USA, 2005. IEEE Computer Society. 20, 29