UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministérial : 7 Août 2006

Présentée par

Rim ABID

Thèse dirigée par **Gwen SALAÜN**
et codirigée par **Noël DE PALMA**

préparée au sein **du Laboratoire d'Informatique de Grenoble (LIG) et INRIA Grenoble Rhône-Alpes**
et de **L'École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Coordination and Reconfiguration of Distributed Cloud Applications

Thèse soutenue publiquement le **16/12/2015**,
devant le jury composé de :

Abstract

Cloud applications consist of a set of interconnected software components running on multiple virtual machines. Thus, there is a need for protocols that can dynamically reconfigure such distributed applications. We present in the first part of this thesis a novel protocol, which can resolve dependencies in distributed cloud applications, by (dis)connecting and starting/stopping components in a particular order. The protocol also supports virtual machine failures. The virtual machines interact through a publish-subscribe communication media and reconfigure themselves upon demand in a decentralised fashion. Designing such protocols is an error-prone task. Therefore, we investigated the use of formal specification languages and verification techniques, in particular the LNT value-passing process algebra to specify the protocol and the model checking tools available in the CADP toolbox to verify it. Moreover, managing distributed cloud applications is a challenging problem because the manual administration is no longer realistic for these complex distributed systems. Thus, autonomic computing is a promising solution for monitoring and updating these applications automatically. This is achieved through the automation of administration functions and the use of control loops, called autonomic managers. Multiple autonomic managers can be deployed in the same system and must make consistent decisions. However, using them without coordination may lead to inconsistencies and error-prone situations. We propose a new approach for coordinating stateful autonomic managers, which relies on a simple coordination language, new techniques for asynchronous controller synthesis and Java code generation. We used our approach for coordinating real-world cloud applications.

Keywords. *Cloud Computing, Dynamic Reconfiguration, Distributed Applications, Fault-Tolerance, Autonomic Management, Coordination, Controller Synthesis, Asynchronous Communication, Formal Verification, Process Algebra Encoding, Labelled Transition System*

Résumé

Les applications reparties dans le nuage sont constituées d'un ensemble de composants logiciels interconnectés et répartis sur plusieurs machines virtuelles. Cet environnement nécessite des protocoles pour reconfigurer dynamiquement ces applications. Nous présentons dans la première partie de cette thèse un nouveau protocole pour résoudre les dépendances dans ces applications. Ce protocole consiste à (dé) connecter et démarrer/arrêter les composants dans un ordre spécifique. Il supporte les pannes des machines virtuelles et les opérations de reconfiguration se terminent toujours avec succès. Ces machines virtuelles interagissent à travers un «publish-subscribe support de communication» et se reconfigurent d'une manière décentralisée. La conception de ces protocoles étant une source d'erreurs. Alors, nous avons étudié l'utilisation des langages et techniques de verification formelles. En particulier, nous avons utilisé LNT pour spécifier le protocole et les outils disponibles dans la boîte à outils CADP pour le vérifier. D'autre part, la gestion des applications reparties dans le nuage est une tâche complexe car l'administration manuelle n'est plus réaliste pour ces systèmes. Nous avons proposé d'automatiser certaines fonctions d'administration en utilisant des boucles de contrôle appelées gestionnaires autonomes. Plusieurs gestionnaires peuvent être déployés pour la gestion de la même application. Cependant, leur utilisation sans coordination peut conduire à des incohérences et à des situations d'erreur. Dans la deuxième partie de cette thèse, nous avons proposé une nouvelle approche pour coordonner plusieurs gestionnaires autonomes. Cette approche repose sur un language de coordination simple, de nouvelles techniques asynchrones pour la synthèse de contrôleur et la génération de code Java. Nous avons appliqué notre approche pour coordonner les applications reparties dans le nuage dans le monde réel.

Mots clés. *Informatique en Nuage, Reconfiguration Dynamique, Applications Distribuées, Tolérance aux Pannes, Gestion Autonome, Coordination, Synthèse de Contrôleurs, Communication Asynchrone, Vérification Formelle, Encodage en Algèbre de Processus*

Contents

List of Figures

List of Tables

Acronyms

A	
ACP	Algebra of Communicating Processes
ADL	Architecture Description Language
B	
BCG	Binary Coded Graphs
C	
CADP	Construction and Analysis of Distributed Processes
CONVECS	Construction of Verified Concurrent Systems
D	
DADL	Distributed Application Description Language
DCS	Discrete Controller Synthesis
DES	Discrete Event System
DSL	Domain Specific Language
E	
ERODS	Efficient and Robust Distributed Systems
F	
FSN	Fonds National pour la Société Numérique
I	
IaaS	Infrastructure as a Service
IT	Information Technology
J	
JVM	Java Virtual Machine
L	
LNT	LOTOS NT
LOTOS	Language Of Temporal Ordering Specification

LTL Linear Temporal Logic

LTS Labelled Transition System

M

MAPE-K Monitor Analyse Plan Execute - Knowledge

MCL Model Checking Language

N

NIST National Institute of Standards and Technology

O

OVF Open Virtualization Format

P

PaaS Platform as a Service

PS Publish-Subscribe Messaging System

Q

QoS Quality of Service

R

RM Reconfiguration Manager

S

SaaS Software as a Service

SLA Service Level Agreements

SmartFrog Smart Framework for Object Group

SVL Scripting Language for Compositional Verification

V

VAMP Virtual Applications Management Platform

VM Virtual Machine

1 Introduction

" Work is the only thing that gives substance to life. "

Albert Einstein

Contents

THis Ph.D. thesis research is part of the OpenCloudware project[1]. We present in this chapter the OpenCloudware project, a brief survey of its expectations, the thesis positioning in the project and, in a second part, we introduce the problem statement, the main contributions, and the thesis organization.

1.1 Industrial Context

1.1.1 Presentation of the OpenCloudware Project

The OpenCloudware project is a collaborative research project. It is funded by the French authorities – specifically through the *Fonds National pour la Société Numérique* (FSN) and started in January 2012 for a duration of three years and nine months. The project, known for its original approaches and technical innovations, is endorsed by *Pôles de compétitivité* Minalogic[2], Systematic[3], and SCS[4].

The project members include 18 partners composed of academic partners (Armines, Inria, IRIT – INP Toulouse, Télécom Paris Tech, Télécom Saint-Etienne, Université Joseph Fourier, and Université de Savoie – LISTIC), industry leaders and innovative technology start-ups (France Telecom, ActiveEon, Bull, eNovance, eXo Platform, Peergreen, Linagora, Thales Communications, Thales Services, and UShareSoft), and a world-renowned open source community (OW2).

Each partner of the OpenCloudware project participated with one or more contributions to the project solution and had its own expectations.

1. http://www.opencloudware.org
2. http://www.minalogic.org/
3. http://www.systematic-paris-region.org/
4. http://www.pole-scs.org/

1.1.2 Project Expectations

The openCloudWare project intends to build an open software engineering platform. This platform will be used for the development of distributed applications, *e.g.,* multi-tier applications such as JavaEE, which need to be deployed on multiple Cloud infrastructures. The project results will be summarized as a set of software components that allow the cloud users to model, develop, and build their applications in order to obtain a multi-IaaS compliant PaaS platform for the deployment, orchestration, performance testing, self-management, and provisioning of the applications. These components have been made available, in October 2015, as open source components through the OW2 open source community.

It is worth noting that the OpenCloudware project addresses multiple technological issues and challenges that we summarize here:

— providing a complete modeling, so-called end-to-end (retro-) modeling, starting from the applications and achieving both PaaS and IaaS services on the platform of cloud computing;
— automating the orchestration of the modules that compose the platform to manage the applications life-cycle in a virtual context;
— reacting to the dynamic changes that can occur in a cloud PaaS application by means of autonomic management, which takes into account elasticity constraints, energy optimization, and also the reliability of the deployed services;
— providing an interoperable execution of multi-cloud IaaS interfaces;
— optimizing the services performance and cost by taking globally coherent decisions;
— ensuring the security in terms of application isolation, encryption, and management.

1.1.3 Position in the OpenClouware Project

The OpenCloudware project contains six technical and scientific subprojects (SP1, 2, 3, 4, 5, and 7), a validation subproject SP6, and a subproject dedicated

to dissemination SP8.
Within this project, the CONVECS and ERODS teams, in which this thesis was prepared, participate essentially in SP1 and SP3. The major goal of SP1 is to define an overall architecture, a model for distributed applications, services of the PaaS platform, and capacities of the IaaS infrastructures. The major goal of SP3 is to develop generic components and tools for the deployment, the automated orchestration, and the performance testing. SP3 also aims at administrating the PaaS platform, particularly the autonomic management of dynamic changes in distributed applications.

The position of this thesis in the OpenCloudware project has responded to the need for the management of dynamic models and the autonomic administration of applications. It also includes the need to formalize the dynamic aspects of these models in order to experiment the administrative functions, to formally specify management protocols for cloud applications, and to verify them using the model-checking and verification tools developed within the CONVECS team.

1.2 Scientific Results of the Thesis

1.2.1 Motivations

During the last few years, the IT operating costs have risen considerably in a continuous way. Therefore, companies have, increasingly, outsourced their IT services and have been reluctant to entrust them to specialists, such as cloud providers. Thus, cloud computing has been an active research topic. In terms of market adoption, it leverages hosting platforms based on virtualization and provides resources and software applications as services over the network. Cloud computing allows, on the one hand, service providers to develop and then sell distributed applications, composed of virtual machines and hosting interconnected software components, worldwide without having to invest up-front in expensive IT infrastructures. On the other hand, it allows the users to access services, to benefit from them through a Web browser, and to pay only

for the services used. As a result, the software systems transform from centralized to distributed and from static to dynamic.
The cloud users need to (re) configure and monitor applications during their time life for elasticity or maintenance purposes. Therefore, after the deployment of these applications, some reconfiguration operations are required for setting up new virtual machines, replicating some of them, destroying or adding virtual machines, handling VM failures, and adding or removing components hosted on a VM. These tasks are required to react to changes (such as the occurrence of a failure), to include new requirements, or to fulfill the users expectations. Some of these tasks are executed in parallel, which involves a high degree of parallelism and complicates their correct execution. Moreover, in the context of the OpenCloudware project, architectural invariants (*e.g.,* started components cannot be connected to stopped components) must be preserved when the reconfiguration operations are executed.
Several research studies and protocols have been proposed to support the self-deployment of cloud applications. These protocols manage static applications, which are composed of a set of virtual machines, ports, components, and connections between the components where the number of these elements and the involved connections are known before the application execution. However, cloud users need protocols that permit to deploy such applications and that are also able to modify these applications during their execution. The existing protocols, *e.g.,* [60, 62, 126] do not support the applications that require to be changed after the deployment phase.

Moreover, today, the requirements of the cloud users are continuously evolving, which leads to the increase of the complexity of the cloud applications and their management becomes a crucial feature. Therefore, the management of these applications is a challenging problem for administrators. The manual administration is no longer realistic for these complex distributed systems and environments. Thus, autonomic computing has become a promising solution for observing, monitoring, and updating these applications automatically. These actions are achieved through the automation of administrative functions and the use of control loops, called autonomic managers. However, available managers can be designed to address the requirements of a particular application domain without coordination. This may lead the whole system to the performance degradation and some undesired situation (*e.g.*, adding two new servers whereas one was enough as a result of a server failure). Therefore, the coordination of multiple control loops managing the same application, particularly distributed cloud applications, is a central issue that appears as a real need in the context of the OpenCloudware project.

1.2.2 Objectives and Main Contributions

One of the goals of the OpenCloudware project is to dynamically reconfigure cloud applications composed of interconnected software components running on different virtual machines in an automatic way. However, achieving dynamically the reconfiguration process is a complicated task. First, when reconfiguring a distributed application, it is necessary to execute the reconfiguration operations, while preserving architectural invariants (*e.g.*, a started component cannot be connected to a stopped one) at each step of the application runtime, particularly when components need to be stopped in sequence across several VMs. Another difficulty with the dynamic reconfiguration of distributed applications is that these applications can be faced with a set of virtual machines or components failures that may occur during the reconfiguration process. Therefore, in this thesis, we take into consideration the fault tolerance, meaning that the reconfiguration process must always terminate successfully even in the presence of a finite number of failures. Another issue, which arises when automating the reconfiguration process, is the necessity to take coherent decisions, which should not lead the system to inconsistent situations.

To automatically bring the deployment and (re)configuration tasks in distributed applications, this thesis contributes with a novel, robust, and fault-tolerant protocol, which enables to execute dynamically reconfiguration tasks, *e.g.*, instantiate and destroy virtual machines and supports virtual machine failures. Moreover, the protocol is decentralized in the sense that all virtual machines communicate information (*e.g.*, binding details, component state) through a publish-subscribe communication media. The protocol also allows to start/stop the disconnected components in a particular order for preserving the application consistency (*i.e.*, architectural invariants) at each step of its execution. This first contribution is assessed through the use of formal methods and verification techniques of the CADP toolbox. Concretely, the evaluation is performed by the specification and the verification of multiple properties on different reconfiguration scenarios and a large set of case studies, ranging from simple ones to other containing several VMs and components.

A second contribution of this thesis is the integration of new controller synthesis techniques in the automatization of administration functions in order to reduce the manual execution of reconfiguration tasks. The automatization process is performed through the use of control loops, called autonomic managers. An autonomic manager observes the application execution, ensures a continuous monitoring, and reacts to events and changes by automatically reconfiguring the applications. This contribution proposes new asynchronous synthesis techniques in order to coordinate several managers and allow them to take coherent decisions globally. The generation of the controller is based on reaction rules and regular expressions used to specify the coordination requirements and interaction constraints. This contribution relies on formal techniques, in particular we use the verification techniques available in the CADP toolbox for generating the controller, validating it, and ensuring that all managers globally satisfy the objectives. We also propose Java code generation techniques that allow to deploy the generated controller for real applications in the cloud.

1.3 Thesis Organization

We have started this dissertation with the industrial context of the thesis, a summary of the scientific motivations, and the main contributions. The remainder of this dissertation is structured as follows:

— **Chapter 2** In this chapter, we present a description of some terms (*e.g.*, distributed applications, cloud computing, dynamic reconfiguration, autonomic computing, coordination, formal methods) used in the results presented in the subsequent chapters.

— **Chapter 3** In this chapter, we present a literature review on dynamic reconfiguration, coordination languages, and controller synthesis techniques.

— **Chapter 4** In this chapter, we present a novel dynamic protocol for reconfiguring distributed cloud applications. The protocol is specified and verified using formal techniques and tools to ensure that it preserves important architectural invariants (*e.g.,* a started component cannot be connected to a stopped component) and satisfies certain properties (*e.g.,* each VM failure is followed by a new creation of that VM).
— **Chapter 5** In this chapter, we describe new asynchronous controller synthesis techniques allowing the generation of a centralized controller, which aims at coordinating several autonomic managers in the cloud. These techniques cover the whole development process from the expression of coordination requirements to the deployment of the controller for coordinating real applications.
— **Chapter 6** In this chapter, we conclude and present some perspectives for further research.

Publications

Some results of this thesis have been published in the following papers:

— R. Abid, G. Salaün, F. Bongiovanni, N. De Palma, Verification of a Dynamic Management Protocol for Cloud Applications, in: Proc. of ATVA'13, Vol. 8172 of LNCS, Springer, 2013, pp. 178–192.
— R. Abid, G. Salaün, N. De Palma, S.M. Kare, Asynchronous Coordination of Stateful Autonomic Managers in the Cloud, in: Proc. of FACS'15, LNCS, Springer, 2015, To appear.

2
Background

" The point of philosophy is to start with something so simple as not to seem worth stating, and to end with something so paradoxical that no one will believe it. "

Bertrand Russell

Contents

In this chapter, we introduce techniques and definitions of some terms used in the results presented in the subsequent chapters. First, we give definitions and general characteristics of distributed applications, cloud computing, dynamic reconfiguration, autonomic computing, and coordination. In a second step, we introduce existing formal techniques and tools that have been employed to achieve the objectives and contributions of the thesis.

2.1 Distributed Applications

Since the appearance of computers in the middle of the 20th century, the application complexity has been continuously and exponentially growing. In fact, applications are more sophisticated and able to respond to new users needs and requirements in terms of Quality of Service (QoS), availability, performance, and also productivity [83, 122]. These applications often consist of a set of interconnected software components that interact together, where configuration attributes characterize each component. All components are hosted on different possibly remote machines.

2.1.1 Characteristics of Distributed Applications

Various types of distributed applications have been developed and used in different domains. All of them are characterized by the following important features [118, 18, 37, 73]: (i) the composition of several entities, which are distributed across multiple machines or servers geographically distant; (ii) the deployment of that entities on heterogeneous servers; (iii) an important diversity of resources used for the decision-making process and for the reduction of the reaction time; and (iv) the ability to take into account dynamic modifications and to inter-operate with other applications deployed in the same environment. These features are fundamental to the understanding of this type of applications. Furthermore, they are accompanied by a set of issues, *e.g.,* a low autonomy, a high administration cost, and a resource under-utilization.
First, the lack of autonomy in the administration of a distributed application has a significant impact on limiting its adaptation to the changes that may oc-

cur in its environment. Actually, the installation phases or application updates require, sometimes, the intervention of an expert to examine the machines used for the application deployment.
Second, each distributed application also requires the intervention of administrators, who need diverse knowledge bases of hardware, middleware, and also of the application itself, in order to configure it properly. Thus, the administration of the distributed applications has become a critical task in terms of time and cost. According to J. Moad [109], A. Eastwood [56], and L. Erlikh [58], the cost of software maintenance and evolution management represents between 75% and 90% of the total budget spent on information technology.
Moreover, taking into account constraints on reliability, security, and quality of service requires that each application is deployed on a set of machines dedicated to it, which leads to a potential under-utilization of the same hardware resources. Referring to a recent study conducted by Kelton Research [120], commissioned in association with the Alliance to Save Energy, the world's largest IT departments have 15% of servers that are not doing anything useful according to 72% of the server managers.

In this context, the OpenCloudware project aims at building an open software platform. The platform will be used for the collaborative development of distributed applications that need to be deployed on multiple Cloud infrastructures. Modifications in distributed applications composed of interconnected components play a significant motivation of this work, with a focus particularly set on the dynamic reconfiguration and how reconfiguration operations can be executed.

2.1.2 Purposes of Distributed Applications

Distributed applications are composed of interconnected software components that interact together to access services and exchange information. To ensure interactions between the different components, five important purposes have to be guaranteed [51]: *(i)* the resource sharing, which presents the ability to easily share data, hardware, and software resources among different users with access control; *(ii)* the scalability, which presents the ability to provide a higher load, especially when adding users, transactions, or data volume; *(iii)* the fault-tolerance, which presents the capacity to minimize the loss of resources when faults happen, while interactions and operations between components must still stand, even when other components have failed; *(iv)* the dynamicity, which presents the ability to, easily, make the system extensible by integrating or removing components during the application life cycle in order to meet new users' requirements; and finally *(v)*, the transparency, which presents the ability to duplicate the resources invisibly and ensure seamless access to both local and remote resources.

These purposes, especially resource sharing, fault-tolerance, and openness play a fundamental role in this thesis. As stated before, one of the major contributions of this thesis is designing a fault-tolerant protocol, which dynamically reconfigures distributed applications by adding or removing components to include new users' requirements.

2.2 Cloud Computing

2.2.1 Cloud Computing Definition

There exist different definitions and concepts that aim at describing cloud computing [34, 45, 129, 140, 131, 65, 106, 71]. We present in this chapter some notions of cloud computing proposed in the literature (Table 2.2.1) to define a common denominator for all these concepts. In [65], J. Geelan introduces some cloud computing definitions, which are proposed by several experts, *e.g.,* P. Gaw, J. Kaplan, K. Sheynkman. According to K. Sheynkman, the deployment, the optimization of energy consumption, and the provisioning of resources on-demand are the main goal of cloud computing. Other experts, such as L.M. Vaquero [129] and the National Institute of Standards and Technology (NIST) [106], consider the cloud as a convenient model, which enables sharing virtualized resources, on-demand, for a large pool of users who pay only what they consume. R. Buyya [34] and L. Wang [131] describe the cloud as a set of networks enabled services, which focus on service-level agreements (SLA) and quality of service

(QoS).

Table 2.1 – Cloud Computing Definitions

Author, Year	Definitions
P. Gaw, 2008 [65]	Cloud computing refers to the bigger picture ... basically the broad concept of using the internet to allow people to access technology-enabled services.
J. Kaplan, 2008 [65]	Cloud computing is a broad array of web-based services aimed at allowing users to obtain a wide range of functional capabilities on a 'pay-as-you-go' basis that previously required tremendous hardware/software investments and professional skills to acquire. Cloud computing is the realization of the earlier ideals of utility computing without the technical complexities or complicated deployment worries.
K. Sheynkman, 2008 [65]	Cloud model initially has focused on making the hardware layer consumable as on-demand compute and storage capacity. This is an important first step, but for companies to harness the power of the cloud, complete application infrastructure needs to be easily configured, deployed, dynamically-scaled and managed in these virtualized hardware environments.
R. Buyya et Al., 2008 [34]	A Cloud is a type of parallel and distributed system consisting of a collection of inter-connected and virtualised computers that are dynamically provisioned and presented as one or more unified computing resources based on service-level agreements established through negotiation between the service provider and consumers.

L.M. Vaquero et Al., 2009 [129]	Clouds are a large pool of easily usable and accessible virtualized resources (such as hardware, development platforms and/or services). These resources can be dynamically reconfigured to adjust to a variable load (scale), allowing also for an optimum resource utilization. This pool of resources is typically exploited by a pay-per-use model in which guarantees are offered by the Infrastructure Provider by means of customized SLAs.
L. Wang et Al., 2010 [131]	A computing Cloud is a set of network enabled services, providing scalable, QoS guaranteed, normally personalized, inexpensive computing infrastructures on demand, which could be accessed in a simple and pervasive way.
NIST, 2011 [106]	Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.
E. Griffith, 2015 [71]	Cloud computing means storing and accessing data and programs over the Internet instead of your computer's hard drive. The cloud is just a metaphor for the Internet. It goes back to the days of flowcharts and presentations that would represent the gigantic server-farm infrastructure of the Internet as nothing but a puffy, white cumulonimbus cloud, accepting connections and doling out information as it floats.

From these definitions and common concepts, we propose the following definition of cloud computing:

Definition 1. *Cloud computing is a set of services and programs accessible over the* *networks from anywhere, at any time, and on various media. It enables users to purchase virtualized resources on-demand, pay only what they consume, and optimize the energy consumption.*

2.2.2 Cloud Characteristics

According to the NIST company [106], cloud computing is composed of five essential characteristics, which are: (1) on demand self-services refers to the ability to consume services (*e.g.,* email, applications, network storage), as needed, without requiring human interaction with the service provider. The consumer pays only what he uses, *i.e.,* only for the amount of the resources he uses. (2) Resource pooling means that computer resources are grouped into a large pool, which allows to serve all the consumers. These resources are dynamically allocated according to the demand of users. The consumer has no control or knowledge over the resources providers and locations. (3) Rapid elasticity refers to the cloud capabilities that can be elastically released to scale outward and inward about the request. For the consumer, the required resources must be appropriated in any quantity and at any time. (4) Broad network access allows the availability of cloud services over the network and their accessibility through standard mechanisms, *e.g.,* mobile phones, smartphone. (5) Measured services mean that cloud systems automatically control, optimize, and monitor the use of resources by using the model "pay only what you consume" which provides the transparency for both service provider and service consumer.

These characteristics explain the need for protocols that are not limited to the self-deployment issues where the application model (virtual machines, components, ports, and bindings between components) to be deployed exists and guides the configuration process. The cloud users need protocols that work, as automatically as possible, in all the situations where changes have to be applied on to a running application. The protocols should dynamically reconfigure the applications. The protocol developed in this thesis dynamically reconfigures and manages cloud applications, which consist of interconnected software components. Every component exports services that it is willing to provide and imports required services. Therefore, each service may be needed by several components and can thus be accessible to several imports.

2.2.3 Cloud Service Models

There are different cloud service delivery models, which are provided to the cloud users and divided into the three main following Layers [106]:

— **Infrastructure as a Service (IaaS)** is the basis layer, which contains all the resources provided to the cloud consumers. IaaS cloud providers offer these resources (*e.g.,* CPU, memory, network) on-demand to allow the consumers to deploy and run their applications. Examples of typical IaaS providers include Amazon EC2[1], Windows Azure[2], Google Compute Engine[3], and Rackspace OpenStack[4].

— **Platform as a Service (PaaS)** is located above the IaaS layer. It contains all the tools and technologies necessary to make the cloud users build, deploy, and manage the lifecycle of their applications. It allows them to focus mainly on their applications development without taking into consideration the configuration of the other infrastructures. Examples of PaaS providers include Salesforce.com[5], Google App Engine[6], CloudBees[7], and Cloud Foundry[8].

— **Software as a Service (SaaS)** is placed above the PaaS layer. It contains a set of applications used by the final consumers. These applications can be directly accessible via the Web browser. Examples of SaaS suppliers include: email providers (*e.g.,* Gmail), office providers (*e.g.,* Google Docs), storage providers (*e.g.,* DropBox). The users of these applications do not have to worry about the other infrastructures such as network and operating systems. However, the developers have to handle their application's configuration. They benefit from the computing power offered by

1. https://aws.amazon.com/fr/ec2/
2. http://azure.microsoft.com/fr-fr/
3. https://cloud.google.com/compute/
4. http://www.rackspace.com/cloud/openstack
5. http://www.salesforce.com
6. https://appengine.google.com
7. https://www.cloudbees.com/
8. https://www.cloudfoundry.org/

the cloud computing.

The protocol that we design aims to dynamically reconfigure distributed cloud applications by instantiating/destroying VMs and adding/removing components in order to include new users' requirements or environment changes. In this context, instantiating virtual machines, which consist of interconnected software components, ensures the execution of these distributed applications on different IaaS layers. Indeed, the core of the system of our protocol implementation is based on an IaaS abstraction layer, such as Amazon EC2.

2.2.4 Cloud Deployment Models

Nowadays, there is an immense growth of cloud adoption in real applications. This growth is explained by the need to, increasingly reduce the control operating costs. However, before choosing and using the deployment models available on the cloud, users must understand their requirements in order to avoid cases where the cloud can bring security risks and challenges for the systems management. The choice of a deployment model is based on the user needs. According to the NIST company [106] and an article published in CloudTweaks.com [49], there exist four primary deployment models in cloud computing, which are the following:

— **Private Cloud** is very popular in business. It brings a large value from the security point of view, but not from the cost efficiency point of view. This model ensures data security by building and maintaining hosting only for specific clients. When there is a security problem, it is resolved through a secure-access VPN. The private model is used by the organizations, which need more control over their infrastructures. Therefore, for a critical application, it is more advisable to use a private cloud. Examples of private cloud providers include HIPAA [3] and Sarbanes-Oxley Act (SOX) [5].

— **Public Cloud** is available to the general public (*e.g.,* business, academic, and government organization). It allows to bring down IT operating cost. The public cloud provides services and infrastructures to various clients. These services may be supplied free of charge or on the pay-per-use basis. Examples of typical infrastructure providers of public cloud include Google, Amazon Elastic Compute Cloud (EC2) [1], and Google App Engine.

— **Hybrid Cloud** is the composition of two or more distinct cloud infrastructures with orchestration between them. It allows, for instance, the cloud users to take advantages of both private and public deployment models. In fact, it helps them to benefit from several applications and data available on a private deployment model, and to to share data and applications on a public cloud.

— **Community Cloud** is shared across several organizations and community members that have common concerns (*e.g.*, policy, security requirements, and compliance considerations). Using this model helps the organizations to reduce the operational cost. Examples of providers of community cloud include GovCloud [2] and NYSE Capital Market Community Platform [96].

Finally, choosing a cloud deployment model is based on several reasons such as business requirements.
In this context, a focus of the OpenCloudware is the development of an open source platform for private, public and also hybrid clouds running across different IaaS Infrastructures.

2.3 Reconfiguration of Distributed Applications

Nowadays, the task of reconfiguring an application (*e.g.*, changing its configuration) is a real burden, especially with distributed applications, which run on different virtual machines and are enabled by scalable cloud infrastructures. The reconfiguration is a fundamental activity for administrators. Indeed, it allows them to adapt a running application by taking into account new requirements and environmental changes. This enables the application to move from a configuration$_i$ to another configuration$_{i+1}$ in some particular situations, such as: *(i)* when an error is detected, faulty components may need to be repaired, other components may need to be removed or stopped, and the system tasks may need to be reassigned; *(ii)* when the physical location of resources or users is constantly changing, the application structure needs to be adapted; *(iii)* when new requirements are added, the application must evolve by the addition of new elements or the removal of existing elements; and *(iv)* when maintenance operations are required, some elements may be temporarily stopped or removed.

In all these cases, reconfiguration is needed in order to respond to new requirements, to reassign and adapt the application structure, or to avoid undesired configurations occurred by the removal or failure of components.

We distinguish two types of reconfiguration:

— **Programmed reconfiguration** refers to the reconfiguration that is anticipated when designing the program, and that will be executed in a running application. Therefore, it is possible to predict the components that will be deployed during the application's lifetime. This kind of reconfiguration requires the shutdown of the application in order to be applied. Programmed reconfiguration is not sufficient to satisfy the unexpected requirements and to react to sudden environmental changes. Therefore, it is not suitable for distributed applications, especially cloud applications.

— **Dynamic reconfiguration** is not known when designing the system. It corresponds to arbitrary changes that may intervene at each step of the application runtime. This reconfiguration is introduced to satisfy new requirements or to react to environmental changes. It is crucial to perform a reconfiguration dynamically on a managed application at runtime, where the components cannot simply be started or shut down. Dynamic reconfiguration allows the resources to be added, deleted or moved when the operating system is running, without deactivating it. This kind of reconfiguration does not need the shutdown or the redeployment of all the system to update it. Therefore, the interruption of services is minimized, system availability is preserved, and so, the system can execute in a continuous way. Dynamic reconfiguration is applied in different domains. This includes, for instance, complex embedded systems used in the air-traffic control systems and automotive industry. At this point, complex embedded systems used in the automotive industry contain software operations that require to be running in different modes. Switching from a mode to another one needs that all components, whose number can be not known a priori, are updated and form a consistent system state. This is a major challenge in the area of dynamic reconfiguration. Readers interested in further detailed descriptions of the utilization of the dynamic reconfiguration within the automotive industry can consult [38].

During our work, we have specifically focused on dynamic reconfiguration, which fits in the area of distributed applications and is required in the context of cloud applications. Our goal was, first, to achieve some reconfiguration operations (*e.g.,* adding a new component, destroying an existing virtual machine, and handling virtual machine failures.) and second, to automatically execute them.
The addition of new elements (*i.e.,* components or virtual machines) does not raise particular problems. Conversely, the removal of existing elements and handling failures may modify the application structure. This can lead to the violation of some architectural invariants (*e.g.,* started components are connected to stopped components). Moreover, some of these operations are executed in parallel, which complicates their correct execution.

2.4 Autonomic Computing

IBM introduced autonomic computing in 2001 [85]. It was inspired by the strategies used by the human nervous system. During the last few years, autonomic computing has been an active research topic. It aims to design software systems and automate as much as possible the self-management of applications and the administration functions. This paradigm is characterized by different features, such as self-healing and self-configuration, and seeks to use automatic management approaches [76, 132]. It was particularly adopted for the self-management of large-scale complex distributed systems, where the management is a challenging problem that cannot be ensured easily in a manual way [85, 134, 46].

2.4.1 Autonomic Manager Definition and Implementation

The main goal of the use of autonomic computing is to reduce error-prone situations and to provide automated and intelligent decisions, especially in the existence of conflicting requests. In this context, the role of the human is to define a set of rules governing the autonomous process instead of controlling manually the systems, which relieve the human from complex tasks.
Autonomic computing is based on the use of autonomic managers [84] that are built as control loops. An autonomic manager adapts and reacts to events and changes detection, such as software failures [85, 87]. Actually, it observes the application execution, ensures a continuous monitoring, and reacts to events and changes detection by automatically reconfiguring the application. Typically, an autonomic manager is implemented using the MAPE-K (Monitor Analyze Plan Execute - Knowledge) reference model [28] (see Fig. 2.1).

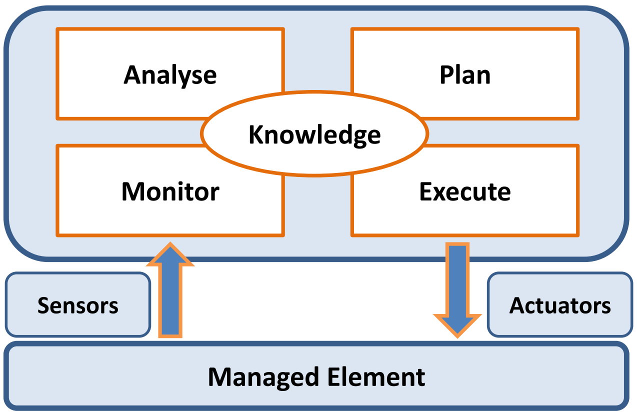


Figure 2.1 – The MAPE-K reference model

The MAPE-K reference model is composed of the following main functions and a knowledge model:

1. **Monitor**: this function monitors the execution of the driven elements using a set of sensors. It receives details and changes from the sensors and

then, generates information (*e.g.,* failure, overload, underload) which are sent to the analyze function.

2. **Analyse**: this function analyzes the information received from the monitor function and then, takes decisions to change or not the managed system. It sends the decision to the plan function.
3. **Plan**: this function, after receiving the decision from the analyze function, generates a reconfiguration program that contains the required reconfiguration operations. The program is transferred to the execute function.
4. **Execute**: this function executes the reconfiguration operations that are applied through a set of actuators on the managed element.
5. **Knowledge**: this model is used by an autonomic manager as a basis for the administrative decisions. It provides information that the autonomic manager may maintain about the managed element.

The increasing complexity of applications implies the use of various and heterogeneous autonomic managers in the same system. Multiple research studies have demonstrated the importance of using autonomic managers. Today, multiple managers are designed to ensure administrative tasks. Examples of autonomic managers include the self-healing and self-protecting managers [112, 33, 103].

2.4.2 Coordinating Multiple Autonomic Managers

Today, using an entirely autonomous system, in which all the administrative functions are ensured, requires the use of multiple autonomic managers. In this context, when several managers monitor the same system, they should necessarily take coherent decisions. Therefore, we must coordinate all the used managers in order to enable them to make coherent self-management decisions. We present in Table 2.4.2 some definitions of coordination proposed in the literature.

Table 2.2 – Coordination Definitions

Author, Year	Definitions
A. H. Bond and L. Gasser, 1988 [24]	Coordination is a property of interaction among some set of agents performing collective activities.
N. Carriero and D. Gelernter, 1989 [67]	Coordination is the process of building programs by gluing together active pieces. Where each active piece means a process, task, data object, agent or application. The main goal of the glue is to allow all independent activities to communicate and to synchronize together. The role of a coordination language is to provide this kind of glue.
T. Malone and K. Crowston, 1994 [100]	Coordination is the process of managing dependencies among activities.
F. Arbab, 1998 [13]	Coordination is the study of the dynamic topologies of interactions among Interaction Machines, and the construction of protocols to realize such topologies that ensure well-behavedness.

From these definitions and common concepts, we define the notion of coordination as follows:

Definition 2. *Coordination is a solution allowing the interaction between several entities, e.g., agents, applications. It allows them to manage dependencies and take globally (i.e., for the system rather than for themselves) coherent and optimal decisions, which avoid performance degradation and system consistency problems, and also limit the energy consumption.*

In [13], F. Arbab distinguishes different characteristics of coordination languages:

1. a coordination language that may be data - or control-oriented, where the coordination is the result of the interaction with the data. In that category, the data handles the activation and deactivation of the control flow;

2. a coordination language that may be endogenous or exogenous, where the endogenous language may provide some concrete primitives necessary for the semantic of computation. Examples of endogenous languages include Linda, introduced by D. Gelernter in [66]. An exogenous language may provide primitives required for the coordination of entities without depending on the computations models. Examples of exogenous languages include Manifold, introduced by F. Arbab, I. Herman, and P. Spilling in [15]

The use of autonomic managers for automating the execution of reconfiguration tasks in the context of this thesis will be presented, in Chapter 4. A part of this chapter also provides a holistic vision of our approach for coordinating stateful autonomic managers in the cloud. This approach is based on reaction rules and regular expressions for describing coordination requirements, and new techniques for asynchronous controller synthesis.

2.5 CADP and LOTOS NT (LNT)

2.5.1 Formal Methods

During the past three decades, several formal methods have been proposed. J. Rushby [123], J. Woodcock [138], and C.L. Heitmeyer [77] define formal methods as follows:

Definition 3. *Formal methods are based on the use of mathematical techniques and formal logic to specify and verify requirements and designs for computational systems (software and hardware) at any part of the program life-cycle (e.g., specification, architecture, implementation, testing).*

A guidebook presented by the NASA agency [82] gives some tips and approaches for a successful application of formal methods for developing high-quality systems at a reasonable cost. The suggested approaches have the role of reducing and optimizing the cost of making computer systems and also improving their quality.

In [113], the authors introduce two important characteristics of formal methods: first, formal methods are based on mathematical notations and objects (*e.g.*, sets, sequences), with the intention of formalizing the system description. Second, formal methods allow to check the consistency of a system description, while verifying whether the properties defined by the analysts meet the system requirements. Indeed, they ensure the detection of defects earlier in the software development life-cycle, which is cheaper than detecting them later (*e.g.*, during the implementation). Finding the defects early helps to validate the correctness of the system description. Some defects can be detected through formal methods and not through conventional design methods [50].

These characteristics show that for safety-critical and high-assurance software systems, it is better to supplement the use formal methods, *e.g.* static analysis, formal proof, and enumerative model checkers. We explain below how these three tools can provide a guarantee that a given design does not contain specific flaws:

— **Static analysis** aims to analyze the source code of a program while taking into account the semantics of the language in which the program is written. Examples of static analysis techniques include abstract interpretation, which is applied to the compilation of programs in order to verify the correctness of optimizations and the certification of programs. Among the static analysis tools, we can mention The Astrée Static Analyzer [52] released in 2001 by *École Normale Supérieure*, and the Polyspace Verifier [57] released in 1999 by Gemplus Research and ID-IMAG Labs.

— **Formal proof** aims to prove that a program satisfies properties using a system of rules. The tools that provide proofs of theorems are called computer proof assistants. Examples of computer proof assistants include the Coq formal proof system [6] and Isabelle [4].

— **Model checker** allows to build an explicit model representing all possible executions of the system that will be verified. Most of the time, it is a concurrent system that contains parallelism. Model checking and equivalence checking are among the main approaches in the field that verify the proper functioning of the system. Model checking is a widespread approach for verifying temporal logic formulas of reactive programs. Equivalence checking is a Rule-Based approach for verifying the equivalence of a concurrent system with another desired model, such as an external behavior. Examples of model checker tools include SPIN [78] which is a generic verification system that supports the design and the verification of asynchronous process systems, SMV [104] which is a symbolic model checker that allows the automatic verification of programs, Uppaal [20] which is a toolbox for verifying real-time systems based on the theory of timed automata, and CADP [64] which is a toolbox for the design of asynchronous concurrent systems.

Among these approaches, we focus on model checking in the subsequent chapters for an important reason. Actually, model checking is able to provide a large set of automated tools reasoning in the programs handled and also in the properties that can be checked on these programs. We focus particularly on CADP. This toolbox is developed by the CONVECS team, in which this thesis was mainly prepared. In the remainder of this chapter, we give an overview of the CADP toolbox (Section 2.5.2) and the syntax and semantics of LOTOS NT (LNT) (Section 2.5.3), which is an input language of CADP.

2.5.2 CADP toolbox

CADP [64] (*Construction and Analysis of Distributed Processes*) is a formal verification toolbox for designing asynchronous concurrent systems (*e.g.,* distributed systems, communication protocols, asynchronous circuits, etc.). An important number of industrial projects and academic institutions utilize the CADP toolbox in several application fields and in many case studies. Various complementary tools developed within the Open/CÆSAR environment have been connected to CADP. The CADP tools can be used to make different analysis, such as model checking, equivalence checking, test case generation, compositional verification, interactive simulation, rapid prototyping, and performance evaluation.

2.5.2.1 Process Algebras

CADP uses process algebras for formally modeling concurrent systems. Process algebras rely on well-defined semantics. It allows us to verify properties of concurrent systems, which communicate and exchange data with synchronization and also, to describe systems execution in terms of Labelled Transition Systems (LTSs). In this context, transitions figuring in the LTSs represent the communication between different processes. Leading examples of process algebras include CCS (Calculus of Communicating Systems) [107, 117], CSP (Theory of Communicating Sequential Processes) [31], and ACP (Algebra of Communicating Processes) [21]. Some languages, such as OCCAM [35], μCRL (micro Common Representation Language) [72], and LOTOS (Language Of Temporal Ordering Specification) [80], combine concepts from process algebraic with features from classical programming languages.
Initially, CADP used LOTOS as main specific language. The recent versions of CADP use LOTOS NT (LNT) [42], which is a simplified and improved version of LOTOS. An automatic translator is developed within the CADP toolbox that ensures the transformation of LNT specifications to LOTOS specifications. Each LOTOS specification can be compiled into a Labelled Transition System (LTS), which enumerates all the possible executions of the corresponding specification. Last but not least, CADP provides scripting languages (*e.g.*, SVL [64]) for describing verification scenarios.

2.5.2.2 CADP Tools

CADP is a rich toolbox, which provides several tools and libraries dedicated to the manipulation of LOTOS and LNT specifications, of Labelled Transitions Systems (explicit [9] and implicit [10]), and of temporal logic formulas. It implements a broad range of optimized state space exploration techniques and verification tools that can be used to make various kinds of analysis [63], *e.g.*, model

9. An explicit LTS usually describes an expected service, where the states and transitions composing it are listed exhaustively.
10. An implicit LTS usually describes a protocol, where states and transitions are given in comprehension.

checking and equivalence checking.

Tools for Labelled Transition Systems

CADP contains several tools dedicated to the manipulation of explicit (such as Bcg graphs) and implicit (explored on the fly) LTSs.
Bcg (Binary Coded Graphs) is a format for the LTSs representation and a software environment for handling this format. It has a central role in CADP. It includes several tools, *e.g.,* Bcg_draw, which gives a graphical representation of Bcg graphs, Bcg_edit, which allows to modify manually the graph generated by Bcg_draw, and Bcg_min, which reduces graphs via strong or branching bisimulation.
Open/CÆSAR is a generic software environment that has the advantages of developing tools, which explore graphs on the fly. Open/CÆSAR plays a central role in CADP. It allows connection to the CADP tools (language-oriented tools and model-oriented tools). Examples of tools written in the Open/CÆSAR framework include Bisimulator (for comparing two LTSs modulo a given equivalence), Reductor (for eliminating tau-transitions/duplicate transitions and determinizing graphs), Evaluator (the on-the-fly model checker of CADP), and Executor (a random execution tool).
The connection between explicit and implicit models is ensured by several compilers, such as CÆSAR.OPEN for models expressed as LOTOS specifications, Bcg_OPEN for models represented as Bcg graphs, EXP.OPEN for models expressed as communicating automata, and SEQ.OPEN for models represented as execution traces.

Tools for equivalence checking

Equivalence checking is a verification technique that consists of comparing the description of a system behavior with the description of its desired external behavior modulo a given equivalence relation [81]. It also targets minimizing explicit graphs. Examples of tools dedicated to equivalence checking include Bcg_min, for minimizing explicit LTS, Bcg_cmp for comparing two explicit LTSs, and Bisimulator for comparing two LTSs modulo a given equivalence.

Tools for model checking

Model checking is a verification technique, which explores all the possible states of a system [95, 17] and checks that the system satisfies a set of properties represented by temporal logic formulas. Classical tools used for model checking include XTL (eXecutable Temporal Language), which explores and queries explicit LTSs presented in the Bcg format, Evaluator3, and recently Evaluator4, which handles formulas described in the temporal logic formalism MCL (Model Checking Language) [102]. MCL presents an extension of alternation-free μ-calculus and is equipped with regular expressions, data-based constructs, and fairness operators. In this thesis, we used MCL to specify two groups of properties: *(1)* those allowing us to verify the protocol that we propose to reconfigure dynamically distributed cloud applications (Chapter 4); and *(2)* those allowing us to verify the controller generated with the new asynchronous synthesis techniques we propose (Chapter 5). We use model checking to verify that these properties are respected.

Tools for Visual checking

Visual checking is a technique that allows the visualization and edition of Bcg graphs. CADP tools dedicated to visual checking include Bcg_draw and Bcg_edit.

Tools for compositional verification

Compositional verification is a technique implemented within CADP and used to fight state explosion when applying enumerative verifications of complex concurrent systems. Examples of tools that support the compositional checking include EXP.OPEN, projector, and SVL (Script Verification Language), which aims at automating the generation, minimization, hiding, and verification of the LTSs.

Tools for LOTOS and LNT

CADP contains several compilers, which allow the transformation of the LOTOS and LNT specifications to other languages and graphs, such as C or finite graphs. Multiple transformations are ensured using lnt2lotos, which translates a source program, described as a set of LNT processes, into either LOTOS.

CÆSAR.ADT translates LOTOS data types into C types and functions.

During this thesis, we have used several tools available in CADP, in particular, tools for Labelled Transition Systems, model checking, and equivalence checking. Last but not least, there exist other tools, which are also available in CADP and used for distributed verification, testing, and performance evaluation.

2.5.3 LOTOS NT (LNT)

LNT is an input language of CADP, which is supported by the LNT.OPEN tool. It is a simplified and improved version of LOTOS. LNT relies on an imperative specification language that makes its writing and understanding much simpler. In 1993, ISO/IEC defined a revised version of the LOTOS standard. Then, the CONVECS team defined a simplified version, called LOTOS NT (LNT).

As stated before, LNT and CADP are chosen in this thesis. This choice is motivated by several reasons. First, LNT is an expressive process language with formal and operational semantics. It combines the asynchronous parallelism inherited from process algebra with the user-friendly syntax inherited from imperative and functional programming languages. Second, LNT is supported by CADP, a toolbox that contains optimized state space exploration techniques and verification tools (see Section 2.5.2).

2.5.3.1 LNT grammar

The behavioural part of the LNT specification language consists of the following constructs: actions with input/output parameters, assignment (:=), sequential composition (;), non-deterministic choice (select), conditional structure (if), breakable loop (loop and break), empty statement (null), and parallel composition (par). Each process defines an alphabet of actions, a list of typed parameters, and a behavior built using the aforementioned operators. Communication is carried out by rendezvous on actions with a bidirectional transmission of multiple values. The parallel composition explicitly declares the set of actions on which processes must synchronize. If the processes evolve independently

from one another (interleaving), this set is empty. The semantics of LNT programs are defined in [41] using Structural Operational Semantics (SOS) rules. We adopt the following notations to present the LNT identifiers: M for modules, T for types, X for variables, F for functions, L for loop labels, B for behaviors, Γ for channels, G for gates, and Π for processes.
We present in Appendix A an excerpt of the LNT process grammar.

Each LNT process is composed of three parts: an alphabet of actions, a set of typed parameters, and a behavior built using the constructs as mentioned earlier. The LNT language provides communication semantics that rely on multiway rendezvous on actions with bidirectional transmissions of value exchange on the same gate. Each action may be followed by a sent value (!) or receive value (?). The parallel composition explicitly declares the set of actions on which two processes or more must synchronize. If the processes evolve independently from one another (interleaving), this set is empty (the "par" is not followed by actions).

For illustration purposes, we give an example (see Example 2.1), which describes the behavior of an automatic hot drink vending machine. The example shows how the machine, if it is not failing, is always waiting to distribute a drink. Once the customer inserts money, he can choose a certain drink (coffee or tea). Then, the machine returns an error message if the client did not insert enough money or delivers the drink and the rest of the money.

2.6 Conclusion

We have presented in this chapter some general background used to support the thesis. The chapter begins with some definitions and general characteristics of distributed applications, cloud computing, dynamic reconfiguration, and autonomic computing. It also presents well-known definitions of coordination proposed in the literature. Finally, we have surveyed existing formal techniques and tools, which motivate the use of the LNT specification language and model checking in order to specify and verify the dynamic reconfiguration of distributed applications and the coordination of stateful autonomic managers

Exemple 2.1 Hot drink vending machine

```
process CoffeeTeeMachine [PIECE_RECEIVED:any, CAFE:any, THE:any,
        ERRORMSG:any, DRINK:any, PIECE_RETURNED:any] is
   var drink_price, rest_money: nat in
      drink_price:=10;
      loop L1 in
         select
            (* machine failure *)
            break L1
         []
            (* money insertion and drink choice *)
            PIECE_RECEIVED (?money of nat);
            select
               CAFE
            []
               THE
            end select;
            if (money > drink_price) then
               (* deliver the drink *)
               DRINK;
               (* compute the rest *)
               rest_money:=money - drink_price;
               only if(rest_money>0) then
                  (* deliver the rest *)
                  PIECE_RETURNED(!rest_money)
               end if
            else
               (* money inserted is not sufficient *)
               ERRORMSG
            end if
         end select
      end loop
   end var
end process
```

in the cloud.

3
State of the Art

" No one wants to learn from mistakes, but we cannot learn enough from successes to go beyond the state of the art. "

Henry Petroski

Contents

We review in the first part of this chapter existing languages and approaches that have been used to reconfigure applications (*e.g.,* distributed applications), including a comparison of the features and limitations of the reviewed approaches. The second part of the chapter overviews some well-known techniques for coordinating entities, including the coordination of autonomic managers. Finally, controller synthesis techniques are surveyed.

3.1 Dynamic Reconfiguration

In the last 20 years, dynamic reconfiguration has been an active research topic with the advent of distributed applications. It has been studied in several research areas that include software architecture [16, 115, 105, 97, 88, 9, 68] (*e.g.,* software architecture in the cloud), graph transformation [133, 8, 90], and reconfiguration patterns [32, 70, 116].
In [88, 97, 99, 98, 9, 133, 36, 124, 90], the authors propose various formal models and languages, such as Darwin and SmartFrog to specify dynamic reconfiguration of component-based systems and distributed applications whose configurations can evolve (addition/removal of components and connections) at run-time. Some of these approaches are based on Architecture Description Languages (ADLs) that have textual and/or graphical syntax used to specify software architectures. In this context, the purpose of an ADL is allowing both developers and clients to reason together in order to provide understandable system abstractions. Multiple ADLs are used in several fields of studies and industries, but few of them offer a description of dynamic software architectures, where dynamic reconfiguration operations are strongly required.
Let us summarize in the following paragraphs the languages and approaches that have been suggested for the specification, deployment, and dynamic reconfiguration of distributed systems and component-based systems. These approaches attempt to address three major challenges, which are: (1) deploying and managing distributed applications, *e.g.,* cloud applications; (2) defining reconfiguration plannings that must be performed; and (3) automating the (inter) connection of software components in order to realize desired target architectures.

We will show that some of these languages and approaches concentrate on one single challenge (*e.g.,* designing a target architecture), while others try to tackle more than one challenge in a unified manner.
Some of these languages and approaches are used in the context of cloud computation, whereas others are used for component-based systems.

3.1.1 Reconfiguration in the Context of Cloud Computing

3.1.1.1 DADL and SmartFrog Languages

In [108], the authors propose DADL (Distributed Application Description Language), a language for managing virtual resources. DADL allows the expression of constraints on resources in terms of service level agreements (SLAs) and elasticity rules when describing an application architecture with its available resources. The DADL language is an extension of the open source SmartFrog project (Smart Framework for Object Groups), which is a Java-based framework, developed at Hewlett-Packard European Research Laboratories in Bristol [69], and tested on client and server operating systems. SmartFrog, written in Java, can be implemented in any operating system that supports Java virtual machines (JVMs). It is used to configure, deploy, manage, and describe the needs of distributed software systems. SmartFrog is based on available resources and declarative descriptions of the components that compose the application. The SmartFrog component model is composed of the following three parts:

— **configuration data** are defined in a uniform way using standardized APIs. It may refer to the components themselves or references to other components. For each configuration data, an initialized phase is defined to specify if it will be initialized statically via the application description or generated dynamically according to the requirements defined at run-time. The SmartFrog component model includes two mechanisms: composition and extension. The composition mechanism allows the use of a component description as an attribute in another one, while the extension mechanism allows the combination of separated components descriptions as an attribute in a reference which does not have a description;

— **a life-cycle manager** uses the configuration data and multiple specific methods to drive a configuration when creating, initializing, starting, stopping a component, and also consulting its state. The SmartFrog allows the description of a system as a group of components. The interrelated components can present software, data, and also hardware. Each component is bound to other components through parent-child relationships. In the same system, different components communicate with each other via the SmartFrog framework. The configuration data of the system members may be inter-related with one life-cycle manager, which emerges from the life-cycles of all the system members (*i.e.,* the components that compose the system). The set of systems is independent, but SmartFrog provides a discovery process that locates them. The life-cycle managers allow to initialize, implement, and monitor the application in a deployed system;

— **a managed entity** presents the component functionality. It uses configuration details to give an interpretation of the component description.

Finally, SmartFrog provides an automated and optimal allocation of cloud resources. It allows its users to build and deploy their running management systems. However, it requires specific components to deploy the system automatically.

Evaluation

SmartFrog supports distributed application deployment and also allows the automation of the component instantiation, initialization, and startup phases, but not the application deployment. It presents some similarities with our approach. However, contrary to us, it does not explain either how a life-cycle manager preserves architectural invariants when stopping a running system. It does not explain either how it manages the component failures. Furthermore, it is not an open source product and has the following limitations:

— the SmartFrog framework requires the use of nodes to install a Java Virtual Machine;

— SmartFrog does not allow to deal with software installation management.

3.1.1.2 RESERVOIR project

In [44], the authors discuss the implications of architecture definition for distributed applications that are deployed on cloud environments. The authors also propose configuration protocols, which are based on Application Description Languages (ADLs). The ADL is used to define a contract, between an IaaS provider and an application developer, in which the IaaS provider can meet some QoS objectives. A defined contract contains a specification of architectural constraints and invariants at runtime used to explain how software architectures must be described. The authors also adopt, for the manifested language, a model-driven approach with extensions of the Essential Meta-Object Facility (EMOF) abstract syntax, which is defined by the initiative Model Driven Architecture (MDA) of the Object Management Group (OMG). The model is used to describe a distributed application, its requirements towards the underlying execution platforms, its architectural constraints (*e.g.,* concerning placement, collocation, and startup/stopping order), elasticity rules that must be declared when deploying an application on the cloud environment, and finally new abstractions (*e.g.,* specification of on-demand scaling). The requirements, constraints, and rules are used to describe when and how the application must consume more/fewer resources from the cloud. In this work, the proposed architecture targets at supporting the abstractions and specifying clear semantic for the ADL, while respecting the architecture using denotational approach. As for the deployment mechanism, components are deployed on the RESERVOIR-based infrastructure, which is provided by the RESERVOIR project [121].

Evaluation

This approach proposes an architectural description that can evolve at runtime. However, these works still have some limitations, which are:

— the approach guarantees the application deployment, but not in a decentralized fashion, which harms the scalability of applications that can be deployed;

— this approach does not explain the order of the component instantiations and the dependency constraints.

3.1.1.3 Engage

A recent paper presents Engage [62], a prototype deployment system developed by the genForma Corp company and MPI-SWS Institute, for managing application stack configurations and dependencies between them. Thus, Engage solves the dependencies by using the MiniSat SAT solver. Furthermore, it uses a high-level modeling language to configure services across machines (physical or virtual machines) according to their dependencies, to deploy components, to describe interactions between components, and to manage their life cycles. Engage presents software or hardware components, which are called resources. These resources are described using a Domain Specific Language (DSL), by means of two parts: a declarative one, which is presented by a resource type, and an implementation part that is represented by a resource driver. The resource type describes a set of ports that specify attributes, which are internal to the configuration of the component, attributes which are internal to configurations of other components on which the component depends on, and attributes which are exported by the component. A resource type allows, also, to generate deployment plans and to verify deployment properties. Last but not least, the resource type defines dependencies on other resources. These dependencies can be classified into three broad types: (1) environment dependencies, which present the other resources that are required by the current one, located on the same machine (physical or virtual machine), and must be installed before its installation and execution. Examples of environment dependencies include the Java programs that need JRE; (2) peer dependencies, which present the resources that must be used, but not necessarily collocated on the same machine containing the current resource. Thus, it can be deployed anywhere; and finally (3) inside dependencies, which present container resources in which the current resource is running.
The Engage workflow provides a universe of available resources defined by the vendor applications. These resources may be required by other users in order to define partial specifications of systems they want to deploy. Each partial specification is fed to Engage to verify properties, *e.g.*, the union of an environment,

peer, and inside dependencies relations is acyclic. From each partial specification, Engage generates a hyper-graph where nodes present the instances of resources and edges present the dependencies between them.
The resource driver is implemented using a specific language. It presents the state machine that installs and manages the life-cycle of the resources. Each resource driver is composed of a set of states and the transitions between these states. Each state machine has, at least, three states that are uninstalled, active, and inactive. Each transition is described as a guarded action $[\uparrow c]$ A or $[\downarrow c]$ A, where c is a Boolean condition, the $\uparrow$ arrow expresses the necessity to satisfy the condition c for all the resources that a given resource depends on, the $\downarrow$ arrow expresses the need to meet the condition c for all the resources that depend on a particular resource, and A presents the action executed when the transition is triggered (see, Fig. 3.1).

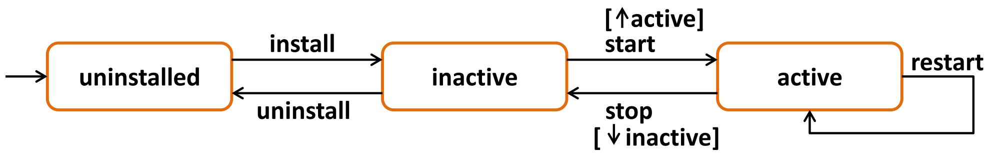


Figure 3.1 – An example of a state machine associated to a resource driver

Evaluation

Despite the compromise provided between efficiency and applicability, Engage presents the following limitations:

— a resource type is used to verify, statically, deployment properties;
— it is not clear how Engage preserves composition consistency and architectural invariants when stopping resource drivers;
— the authors write that going from an active state to an inactive state requires, as a precondition, that dependencies are inactive without explaining the mechanism used to deactivate them.

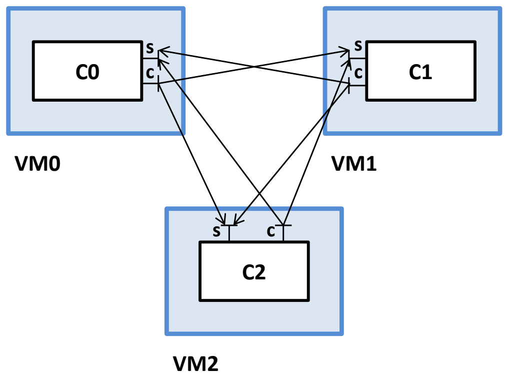
3.1.1.4 VAMP

In [59, 60, 126, 61], the authors describe a general solution, called Virtual Applications Management Platform (VAMP), that automates the configuration of distributed applications in the cloud environments. This solution is based on a representation of distributed architectures. The framework VAMP extends the OVF (Open Virtualization Format) language, which is provided by the Distributed Management Task Force (DMTF) and dedicated to uniform presentations of applications running on virtual machines, with an ADL that describes the global architecture of distributed applications. This extension provides an explicit specification of the component building, interfaces, dependencies, and binding between components. We present in Figure 3.2 an example of an application (left) and the extended OVF descriptor that corresponds to that application (right). The application is composed of three interconnected components, hosted on distinct virtual machines and called C0, C1, and C2, respectively. Each component has a server interface *s* with a set of client interfaces *c* and is connected to other components through bindings among its client interfaces and their service interfaces.

This solution also presents a decentralized and self-configuration deployment protocol that can be implemented on a compensation-based mechanism. In [61], the authors present a reliable self-deployment protocol, which represents an extension of the original deployment protocol VAMP [59]. This protocol can automatically deploy cloud applications in a decentralized fashion, even in the case of a finite number of virtual machine failures. Thus, the protocol supports the failures by detecting and repairing them, but it is not compatible with some reconfiguration features.

Evaluation

This solution introduces a protocol for deploying distributed applications. The protocol allows the automation of the application deployment, but it presents some limitations, which are:



```xml
<!-- Applicative architecture -->
<AppArchitectureSection>
  <definition name="application_name">
    <component name="C0">
      <interface name="c" role="client" .../>
      <interface name="s" role="server" .../>
      ...
      <virtual-node name="VM0"/>
    </component>
    <component name="C1">
      ...
    </component>
    <component name="C2">
      ...
    </component>
    <binding client="C0.c" server="C1.s" />
    <binding client="C0.c" server="C2.s" />
    ...
  </definition>
</AppArchitectureSection>
```

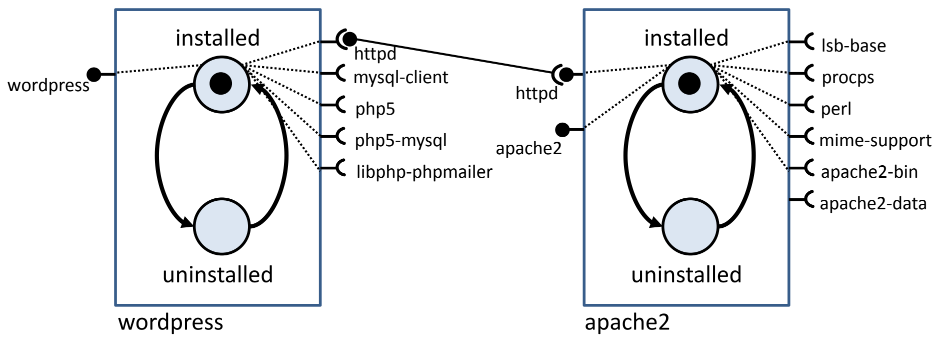
Figure 3.2 – An example of an application (left) and its extended OVF descriptor (right)

— in the applications, all elements are known from the beginning (*e.g.*, numbers of VMs and components, bindings between components);

— the protocol does not allow to modify the application at run-time.

3.1.1.5 Aeolus Component Model

In [91, 40], the authors present a design and an implementation of an approach used for the automatic synthesis of deployment plans. This approach provides a reconfiguration algorithm for deploying heterogeneous software components and computing a sequence of actions to deploy the desired configuration. This algorithm works in the presence of circular dependencies among components. To master the complexity of cloud applications, the authors propose the Aeolus component model and explain how some activities, such as deployment, reconfiguration, and management phases of such applications (*i.e.*, cloud applications), can be automated in this model. Aeolus uses a set of available components, where each component is regarded as a gray-box that owns internal states and mechanisms to change its state during the three phases. A component may also provide and require ports, where each port can be active or inactive according to the current internal state. The Aeolus component model allows to describe several characteristics of components, *e.g.*, the dependencies between components, the conflicts, and the non-functional requirements. Aeolus takes as inputs high-level application designs, user needs, and constraints (*e.g.*, the number of requiring ports which can be bound to a given provided port) to provide valid configurations of target cloud environment, which is composed of a set of interconnected components where each component must require or/and provide services. We present in Figure 3.3 an example of an application model described by the Aeolus model. The application consists of two packages. The first one (wordpress) provides a service named wordpress and requires five services. One of these services is named httpd and supplied by the second package (apache2), which also requires six services.



Package : wordpress
Version : 3.0.5 + dfsg - 0 + squeeze 1
Depends : httpd , mysql-client, php5, php5-mysql, libphp-phpmailer,

Figure 3.3 – An example of packages, services, and bindings described by the Aeolus model

Last but not least, to efficiently deploy and configure their components, the Aeolus users may use a planning tool [92], which supports the deployment of cloud applications or the Zephyrus tool, which allows computing a configuration in order to check if it satisfies a high-level specification. Some of the packages are popularized by Free and Open Source Software (FOSS) distributions, *e.g.*, Debian.

Evaluation

The Aeolus component model focus on mastering the complexity of cloud applications. It automates as much as possible the management of such applications. A component is a grey-box presenting internal states and mechanisms to change its state during the deployment and reconfiguration process. These works present some similarities with ours, but present the following limitations:

— these works do not propose solutions for verifying that all the constraints defined by the components in the initial configuration are also satisfied in the target configurations;

— the Aeolus model takes as inputs the initial configuration, the target configuration, and the required actions. Thus, it does not support modifications that can take place at runtime.

3.1.2 Reconfiguration in the Context of Component-Based Systems

3.1.2.1 Synergy

In another approach [26, 25], the authors propose a robust reconfiguration protocol for an architectural assembly of software components. The protocol takes as input a current assembly, a target assembly, and a set of reconfiguration operations. It transforms the current assembly to the target one by applying reconfiguration steps (*e.g.*, connecting/disconnecting port and changing component states) to a set of connected components, with respecting a reconfiguration contract given to the developers of components. The protocol preserves some architectural invariants and a specified reconfiguration grammar, even when multiple failures occur during the reconfiguration, which is characterized by two phases (see Fig 3.4):

1. the "prepare" phase uses a copy of the current architecture to define the target architecture;

2. the "commit" phase uses the execution of reconfiguration operations by the reconfiguration protocol to evolve the managed system and obtain the target architecture.

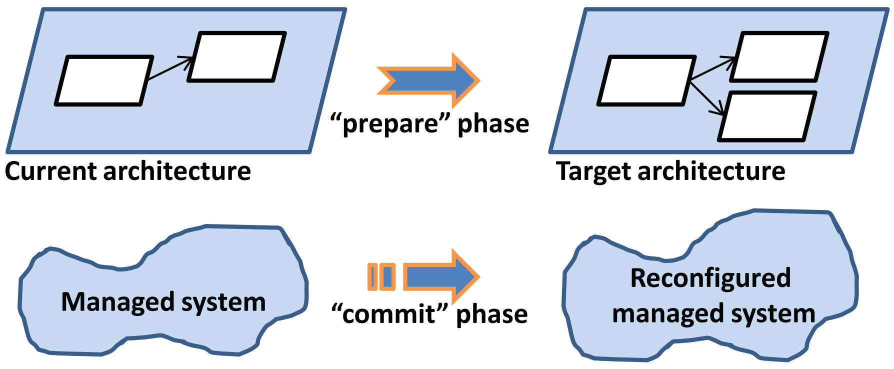


Figure 3.4 – The reconfiguration steps: the prepare and commit phases

Evaluation

This protocol is fault tolerant. It was formalized and proved using the Coq Proof Assistant. The protocol also guarantees that all the reconfiguration steps preserve architectural invariants. However, it assumes that the interconnected components are hosted on the same VM. Thus, it does not consider its distribution across several VMs. Therefore, the protocol does not easily scale to cloud applications that are distributed.

3.1.2.2 Wright

In [11], the authors present another ADL, named Wright. A Wright specification is based on the following elements:

— **the components** are regarded as computational entities. Each component is composed of an interface (input/output ports) and computation, which presents the specified behavior;

- **the connectors** provide the communication between components. Each connector is composed of a set of roles and a glue. Each role defines a behavior expected from one of the components connected to it, while the glue defines the behavior of the connector. Both the expected behavior and the behavior of the connector are specified using communicating sequential processes (CSP). The CSP is also used to specify the computation of basic components;
- **a configuration** presents the design of the system architecture. It is composed of a set of instances of components, connectors, and description of the connection between the component ports and connector roles. The configuration can be also used, in some cases, to replace the specification of computation of complex components.

Dynamic Wright [10], based on the concept of event-based control, was introduced to express architectural reconfiguration that cannot be described through Wright. Dynamic Wright uses a reconfiguration controller to define the states where changes can occur in the system and to trigger reconfiguration tasks. It uses CSP extended with new operators (*e.g.*, del, attach, new) to describe the reconfiguration tasks.

Evaluation

Despite the extension of Wright to be able to describe architectural configuration, it still has some drawbacks:

- the configuration described using CSP must be known at the design time;
- the reconfiguration controller must know all the elements in the architecture;
- changing a component by another one, with different events, may provide a new configuration different from that described using CSP, which may produce reconfiguration problems (*e.g.*, how do connect components in the new configuration, which is not described using CSP).

3.1.2.3 Darwin and Tracta

In [97], the authors describe the system configurations in terms of components and interactions between them using Darwin, which is a pioneer configuration description language that has both textual and graphical notations. This ADL allows to specify the components and the services they require and/or provide. One of the interests of Darwin is taking into account the dynamicity of a configuration topology (*e.g.,* create/delete components, bind/unbind connections) in order to form composite components. In [88, 99, 68, 98] for instance, the authors show how to formally analyze behavioral models of components by means of Labeled Transitions Systems (LTSs) [68] and process algebra (FSP) [98]. In [88, 99], the authors focus on behavioral specifications with components and give the example of a gas station. They use these specifications to analyze the overall system architecture and check safety and liveness properties (*e.g.,* the amount of gas delivered must be equivalent to the quantity paid for). In [68], the authors present an overview of the Tracta project, which gives an extension of the Darwin approach. Tracta checks two categories of properties: (1) safety properties, which are expressed using deterministic LTSs to describe the desired system; (2) liveness properties, which are specified in Linear Temporal Logic. In [99], the models described previously with Darwin, are described textually as finite state processes (FSP), displayed, and analyzed by the Labelled Transition System Analyser (LTSA).

Evaluation

The studies presented in this section present the Darwin language and the Tracta project used for the description of the system configuration and the verification of safety and liveness properties. However, these studies use Darwin, which is limited to simple structural modeling. Moreover, the execution of the reconfiguration operations ensured by Engage (*e.g.,* create a component) are not automated. Our focus is different here because we work on a protocol whose goal is to execute automatically the reconfiguration tasks.

3.1.3 Summary

We have presented in this section a review of different languages and approaches in the field of dynamic reconfiguration. We observe that not all of them are suitable for the dynamic reconfiguration of distributed cloud applications.

Table 3.1 – Comparison between the languages and approaches used for dynamic reconfiguration

	Distributed Applications	Architectural Invariants	Fault Tolerance	Dynamism	Automation
DADL/ SmartFrog	✓	✗	✗	✓	✓
RESERVOIR	✓	✗	✗	✓	✓
Engage	✓	✗	✗	✗	✓
VAMP	✓	✓	✓	✗	✓
Aeolus	✓	✓	✗	✗	✗
Synergy	✗	✓	✓	✗	✓
Wright	✗	✗	✓	✗	✗
Darwin/ Tracta	✓	✗	✓	✗	✗
Our protocol	✓	✓	✓	✓	✓

Table 3.1.3 presents multiple categories of solutions: *(i)* those, such as Synergy, which do not scale to distributed applications; *(ii)* those, such as Engage, which do not propose methods or do not explain how they preserve architectural invariants; *(iii)* those, such as DADL and SmartFrog, which are not fault-tolerant in the sense that they are not able to detect and repair VM failures; *(iv)* those, such as VAMP, which work fine only with specific applications and do not support modifications that can occur at run-time; and finally *(v)* those, such as SmartFrog, which do not execute operations that are completely autonomous.
As stated before, one of our most substantial contributions was to design a protocol that aims to deploy automatically and dynamically (re)configure distributed cloud applications and support VM failures. The protocol must also preserve important architectural invariants (*e.g.*, a started component cannot be connected to a stopped one) at each step of the protocol application. Another goal identified in this thesis is the verification of the protocol using the CADP model checker.

3.2 Coordination Models and Languages

Coordination models and languages provide language support for controlling and communicating software architectures composed of interconnected components. They can be classified into several categories, according to the techniques used to coordinate entities: *(1)* those, which use data available in a common data space, *(2)* those, which use events occurring within the ports of the coordination entities, *(3)* those, which use rules and decision policy, *(4)* those, which use utility functions, and finally *(5)* those, which use shared knowledge.

3.2.1 Data-driven Coordination Models and Languages

In the data-driven coordination model, the coordination media is defined as a shared and common data space, so-called coordination medium, where the values of data being sent and received by coordination entities are useful for defining system states. Indeed, the coordination entities communicate among themselves by exchanging these values. Each entity may deposit (consume, resp.) information at (from, resp.) the coordination medium (see Figure 3.5, left). The data-driven coordination model provides information about how the data should be deposited and consumed into/from the coordination medium, which does not have any idea about their state changes. The focus of these models and languages is on the exchanged information and not on the control flow between coordination entities.

3.2.1.1 Linda

The Linda coordination language allows the separation of the computation part of an application from its coordination part. It is based on simple coordination primitives useful for designing distributed systems. Linda is intrinsically regarded as multi-paradigm by providing generative communication

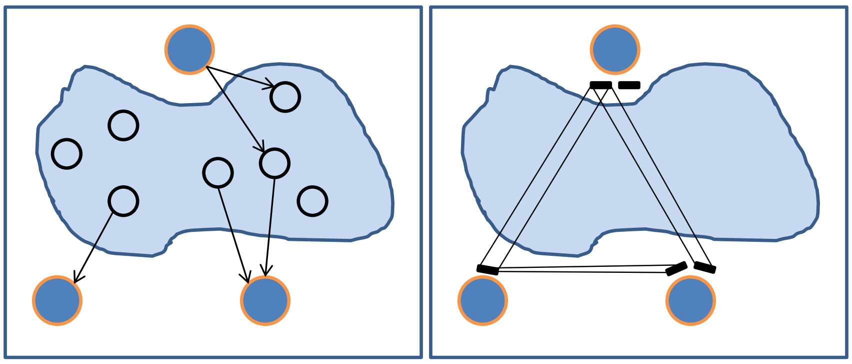


Figure 3.5 – The coordination medium when: using a data driven coordination language (left); using an event-driven coordination language (right)

paradigms, where processes communicate asynchronously by inserting and retrieving data objects, called tuples. The tuples are accessible to all the processes but are bound to none. They are deposited by the sender processes in a shared and common memory called tuple space, from which the receiver processes consume them. This mechanism is decoupled in space, time, and destination (*i.e.,* does not require that a sender (receiver, resp.) process identifies the receiver (sender, resp.) one). Each tuple in the Linda model is composed of a list of information items that are usually of the following form: $''tag'', value_1, ..., value_n$, where the tag parameter allows to distinguish between tuples, a tuple may have 0 or more values, and every $value_i$ represents a target (address) or a communication (data). An extension of a tuple, named template, allows to distinguish its passive values from actual, and formal values, which represent wildcards for matching.
Linda uses four basic coordination primitives for manipulating the tuple space (*i.e.,* operations about how the tuples should be deposited and consumed to/from the tuple space): the `out()` primitives are used when the tuple has been filed in the tuple space; the `in()` and `rd()` primitives are used to find a tuple that matches their template. More precisely, the `in()` primitives are required when a receiver process has retrieved the matching tuple from the tuple space, whereas the `rd()` primitives are required when a receiver process has only examined and read the tuple from the tuple space without modifying and

removing it. When more than one tuple matches the template, only one is selected non-deterministically. Contrariwise, if there is no tuple to match the template, the process is not blocked. The `eval()` primitives are used when creating an active tuple that turns into a passive one after its evaluation. The `in()` and `rd()` primitives are synchronous, whereas the `out()` and `eval()` primitives are asynchronous.

Evaluation

The Linda coordination language is an intuitive language where all tuples are accessed in a concurrently way by multiple processes, but it suffers from the following limitations:

— the coordination medium, so-called tuple space, has a fixed behavior. Thereby, the coordination mechanisms cannot represent generic coordination policies;

— the coordination medium is implemented as a shared and common memory. Linda does not propose other solutions to implement it, especially in the absence of shared memories;

— the Linda model does not support the process failures (*i.e.,* it is not fault-tolerant);

In order to add more coordination features to coordinate more complex applications, the Linda coordination model has been extended, by introducing new primitives and several data spaces. In this regard, examples of extensions of the Linda paradigm include Objective Linda [86], PoliS [47], Shared Prolog [30], and TSpaces [139].

Other examples of data-driven coordination languages and models include Concurrent Constraint Programming [29] and Opus [43].

3.2.2 Control- and Event-Driven Coordination Models and Languages

In the control driven coordination model, the coordination entities interact with the external environment through events occurring on input/output ports. The control driven coordination model allows the coordination medium to observe the states update and the events occurring within the ports of the coordination entities (see Figure 3.5, right). The coordination medium provides information, called coordination laws, about how the events should propagate. The event-driven coordination models and languages focus on the control flow management and not on the information exchanged between coordination entities.

3.2.2.1 Manifold

In [15], the authors present Manifold, a strongly typed and exogenous control- and event-driven coordination language based on the IWIM (Idealized Worker Idealized Manager) model [12]. In the Manifold model, a set of features that include composition and separation of concerns are introduced to manage asynchronous interactions between concurrent computations. Furthermore, the system is defined, topologically, by channeling components through their external behavior, which presents the interfaces that a component may publish. Manifold defines four categories of primitives, which are processes, events, streams, and ports.

Processes

The processes have a set of input/output ports used to send and receive data and are classified into two types: (1) the atomic processes can consume input data, compute, and produce output data through their ports. They are also able to receive and generate events; (2) the coordination processes, called manifolds, coordinate the activities of some atomic processes. Each manifold is composed of two main parts:

— the head contains details, such as the process name and the names of its ports;

— the body is represented as a block that contains a finite number of states. Each state, in turn, is composed of a label that defines the condition allowing transitions to that state and a body that contains all the instructions performed, in a non-deterministic order, upon transition to another state. The main role of a state body is to generate event occurrences.

Events

The events are raised by the components to enact a configuration. In Manifold, a configuration is defined by connections, disconnections, and rearrangement of the binding established between the components, which is necessary for the context of dynamic applications. The manifold model also provides other processes, named observer processes, and are in charge of observing event occurrences. Once an event occurrence is observed by an observer process, it will be put into its event memory. An observed event may be reacted by only one observer process. The Manifold language provides an event-driven state transition mechanism as the only control mechanism.

Streams

The streams are communication links, connected to the ports, and used to transfer data in the form of sequences of bits from a source to its sink. There exist four basic types of stream, which are: BB (break-break) [1], BK (break-keep) [2], KB (keep-break) [3], and kk (keep-keep) [4].

Ports

The ports are used by the processes to exchange information. Indeed, each process has a set of ports located at its boundary. Manifold provides three standard ports (`input`, `output`, and `error`) defined for all instances of the process.

3.2.2.2 Reo

A channel-based exogenous coordination language, called Reo, was designed by F. Arbab in [14] as an extension of the Manifold language. Reo allows to coordinate multiple entities, with supporting a high degree of synchrony. The entities may refer to processes, threads, agents, and software components. These entities, called component instances, execute at one or more locations.
In the Reo model, each component is defined as black boxes and may have in-

1. the BB stream is used to break automatically a connection
2. the BK stream is used to break the source connection and keep the sink connection
3. the BK stream acts as the other way as the KB stream
4. the kk stream is used to never disconnect

put/output ports. The communication between the components is exogenous and established by a set of so-called connectors that coordinate their activities. Each connector may build up using a set of simpler connectors, which are composed of channels and nodes. A channel allows to flow data items by disposing of two types of ends: source end for accepting data into the channel and sink end for putting data out of the channel. A channel in the Reo model can have only one of the two ends or both source and sink ends. Furthermore, Reo disposes of an open-ended number of channels, which allow developers to use their custom semantics to integrate their own channels. Examples of Reo channels include `Sync` channel, `LossySync` channel, `SyncDrain` channel, and `FIFO` channel, which is an asynchronous channel. The Nodes are based on channel ends and classified into three types: source, sink, or mixed, depending on the channel ends. The two first types (*e.g.,* source and sink nodes) allow the interaction of the channel with its environment. The Reo semantics is given by constraint automata, where the transitions are labeled with synchronous ports and, if needed, data constraints on these ports.
In [89], C. Krause presents some reconfiguration tasks in the context of Reo connectors. In this work, the author supposes that both components and coordination layer are distributed. Thereby, the simpler connectors are hosted at different locations (logical or physical machines). Furthermore, these connectors are supposed to connect to each other after knowing when and in which way they will be reconfigured.

3.2.2.3 Evaluation

Reo is a highly exogenous and a powerful coordination language that presents an extension of the Manifold language. The Reo model is composed of a set of components that are regarded as black boxes. It is also based on a set of constraints that may be defined by channels and describe the connectors behaviors. Our focus is different here because we take into account the behavior of the coordinated entities, which execute the reconfiguration process and are modeled as LTSs. We use coordination requirements to describe the behavior of a generated controller, which aims to orchestrate these entities and communicate with them asynchronously.

3.2.3 Coordination Using Rule Agents and Decision Policies

3.2.3.1 Accord

In [93], the authors present the framework Accord, a component-based model designed to build autonomous applications. These applications are based on autonomous components that incorporate rule agents with autonomic properties. Accord allows to coordinate several autonomic components using high-level rules for their dynamic composition. The components are hosted on distributed applications, which may include dynamism in the component states and provided/required services. An Accord component is composed of the following three elements:

- — a functional port that describes the operations provided/required by the component;
- — a control port that controls the system. Indeed, it uses a set of sensors to monitor the component state, a set of actuators to modify the component state and adapt it dynamically, and a set of constraints;
- — an operational port that adds new rules, manages, and removes existing rules dynamically.

The rule agents configure, monitor, adapt the managed components, and ensure the communication between them. To do this, they execute two types of rules: behavior rules, which are used to describe the dynamic composition of the components (*i.e.,* the component configuration) and the behavior of their self-management, and interaction rules, which are used, with functional ports, to coordinate the interaction relationships between the components.

The dynamic composition of an Accord system is based on multiple constraints. This includes, for instance, the composed components that interact together, how components interact together, and when components interact together. The first constraint, *i.e.,* composed components, is described by means of a workflow graph where multiple reconfiguration tasks can be dynamically executed, *e.g.,* adding, removing, and updating components or establishing, removing, and changing interactions in the workflows. This constraint is described using behavior rules, while the second and third constraints are described using interaction rules.

The conflicting decisions between agents are solved with associating priorities with rules. Indeed, a rule that has a low priority should wait for locks held by those having higher priorities. Moreover, several decision policies are handled by the agents in order to monitor the managed components. Agents are classified into multiple categories, such as configuration, context, and composition agents. Accord uses composition agents to support the centralized coordination of workflows adaptation. It also supports the definition of decentralized coordination, which is described using Rudder, but it does not provide any decentralized coordination model.

Evaluation

The Accord model is a component-based model based on rules and decision policies and used to design autonomous applications in distributed environments. However, we identified some limitations which are:

— Accord is based on interaction rules and not on connectors to describe the interaction between components, which prevent components taking independent decisions;

— Accord uses a manual administration to evaluate and update the interaction rules.

3.2.4 Coordination Using Utility-Functions

In [53], the authors present an approach based on multi-agent systems, which are autonomous decision makers that interact with each other to take coherent decisions. The suggested approach provides a consistent control policy for managing the agents (see Fig. 3.6). Furthermore, it introduces four major participants: (1) the performance agents manage the performance by distributing load among the active servers, while taking into account the QoS; (2) the power agents, based on the Energy Management Tool (EMT) program, manage the energy consumption of servers by adjusting the power of the servers; (3) the coordination agents, implemented in Java and hosted on a separate server, have two main roles. It coordinates the two first agents (*i.e.,* performance agents and power agents) and also provides coherent control policies, such as power-performance policy in order to manage a data center, while using a system model and a set of predefined utility functions. Indeed, the coordinating agents receive information describing the QoS and energy consumption (*e.g.*, control signals, monitored performance, and power data). Then, based on these functions, they send decisions about adding or retrieving applications to the performance or power agents. Finally, (4) the policy generators use predefined utility functions and a system model to derive the control policy, such as the power-performance policy, received from the coordination agents and transferred to other agents.

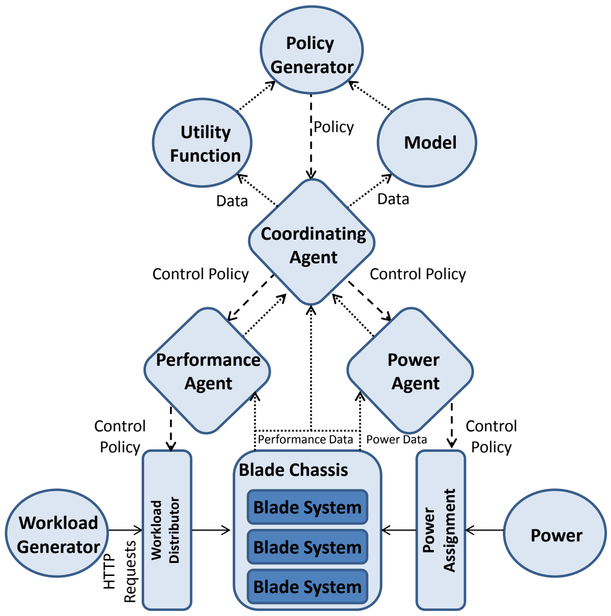


Figure 3.6 – Multi-agent architecture for establishing an energy consumption policy

In turn, predefined utility functions [130] present high-level objectives for managing the behavior of autonomic systems. To do this, the suggested approach defines two types of policies:

— action policies are presented in the form of "IF (condition) THEN action". Therefore, an action policy guarantees that some actions should be executed by the system, once the condition is satisfied;

— goal policies define the high-level objectives of the system. They are translated, using utility functions, into specific actions to the possible conditions. To establish a translation, the utility functions allow to map a current system state to another preferred state.

Evaluation

We observe that this approach, based on multi-agent systems used as autonomous decision makers, focuses on a specific application domain and its results are difficult reusable in other contexts.

3.2.5 Coordination Using Shared Knowledge

In [111], the authors propose a generic integration model that focuses, first, on categorizing autonomic loops in terms of reciprocal interference using their reciprocal interferences. The control loops are implemented using the MAPE-K reference model (see Fig. 2.4.1). This generic model, based on actions and events of inter-loop, can be used to manage the synchronization and coordination of multiple control loops. The control loops communicate with one another through a shared knowledge-based synchronization pattern. This knowledge is composed of two parts:

— private knowledge, which contains necessary internal information asked by a set of internal control loop functions;

— public knowledge, which provides information shared among control loops. When one of these loops needs to modify any information, this knowledge must be synchronized. When multiple loops need to modify the same information at the same time, error-prone situations may occur. To avoid this problem, the authors propose that only the owner of the public knowledge should modify it directly, by using a control loop coordination protocol.

Before explaining the control loop coordination protocol proposed by this approach, let us clarify the difference between an event and an action used by the integration model.

— an action can be used in three cases: *(i)* when executing a decision on the managed system, *(ii)* when notifying another control loop (*i.e.*, inter-loop), and *(iii)* when launching a step within a control loop (*i.e.,* intra-loop). In the two first cases, the action is executed by the execute function (the fourth function of the MAPE reference model). Contrariwise, in the third case, it may be executed by the monitor function to launch the analyze function, or by the analyze function to launch the plan function, or by the plan function to launch the execute function. An intra-loop action is used to change public knowledge or to invoke handler actions.

— an event has one of the two major types: endogenous event that has the managed system as a source and exogenous event that has another control loop as a source.

This approach proposes a coordination protocol, which aims to coordinate multiple control loops. To do this, the execute function of a control loop asks every loop, with which it must take globally coherent decisions, services before executing. Then, it waits for the answers. This mechanism is ensured by triggering an inter-loop event, which will be detected by the monitor function of the second control loop, in order to start it. At the same time, it creates a handler that has all the actions needed to be executed after the reception of the answers. Once the second control loop makes its responses, an inter-loop event will be sent to the first control loop.

Evaluation

The suggested protocol was applied to a scenario in the context of cloud computing. The scenario is composed of multiple self-adaptive applications that communicate with a shared self-adaptive infrastructure. Indeed, the primary role of the coordination protocol in this context is, on the one hand, maximize the Quality of Service at the applications while minimizing costs, and on the other hand to optimize the utilization rate at the common infrastructure. The model was evaluated through simulation-based experiments. However, this approach still has some limitations, which are:

— the model is restricted to a controller loop. It is not clear if the same model can be used for other entities, such as agents, processes, and software components;

— the decisions taken by every control loop must take into account information provided by other loops. Therefore, when several loops control the same application, every one of these loops cannot execute before asking services from the other loops and waiting for all their answers. This slows the response time of the loop;

— in the cloud computing scenarios, multiple unexpected events can occur at each time of the runtime. This approach does not explain how the shared knowledge takes into consideration the changes that have occurred.

Our focus is quite different here because we work on controller synthesis techniques whose goal is to control other entities where components are modeled as LTSs and communicate asynchronously. Moreover, our techniques are based on a set of coordination requirements that take into account all the changes that can occur in the system.

3.3 Controller Synthesis Techniques

3.3.1 Original Theory for Controller Synthesis Techniques

We start this section with a short overview of the original theory for controller synthesis. The approach was originally introduced by Ramadge and Wonham in 1987 [137, 119] in order to resolve problems of Discrete Event Systems (DESs), which contains discrete states, where the controller switches between them by means of events. The approach is called Discrete Controller Synthesis (DCS) and aims to control the logical behavior of discrete event systems originally described using programming languages for programmable logic controllers (PLCs).

The controller synthesis technique focuses on models of entities to be controlled and a set of properties to be satisfied by the models to design a controller. The controller has an impact on the system behavior in order to avoid incorrect behavior specified by the properties. To do this, the controller observes all the system events and enforces it to respect the properties, which are regarded as constraints imposed by the system. They can describe the relationship between states in the models or incorporate temporal aspects. In the context of DESs, the controller synthesis is applied to ensure the properties verifying the system state. The controller must take coherent decisions, even in the context of complex systems (*e.g.*, decentralized systems, system composed of multiple subsystems deployed on distant sites) or in the presence of structural constraints (*e.g.*, communication constraints).

In [137], the authors present a controllable language as a solution for the supervisory of hybrid control systems. This work gives an approach that generates controllers from a particular system called plant and designed as a finite automaton.

[119] proposes a supervisor synthesis algorithm, which allows the automatic generation of a controller (represented by a language C) from a given plant, modeled by a language P as finite state machines, and a specification language. The language P defines an alphabet of events. The generated controller, called maximally permissive, may dynamically enable/disable some events, named controllable events. It also permits all possible legal executions and must satisfy the properties (*e.g.*, safety and liveness properties) and minimal restrictiveness objectives required in the specification. This suggested synthesis approach is based on a classical two-person game approach which: (1) formally synthesize a controller from a representation of all possible behaviors of the system represented as state machines or Petri nets, or STS. (2) Then, check if the monitored system verifies given properties that may describe situations to reach or/and situations to avoid. These approaches can be characterized as restrictive because they directly influence and impact the controlled system. The events issued by the system are broadly divided into two categories: controllable events, which may be prevented by the controller, and non-controllable events, which cannot be disabled by the controller. For a detailed overview of supervisory control theory, the reader is referred to [39, 135].

This synthesis controller approach is based on the supervisor construction concept and regarded as a modular synthesis technique, which decomposes a complex discrete event system (DES) to a set of independent components, defines algebra of the DES and also of the corresponding supervisors. Therefore, the desired supervisor is synthesized as a product system. The interaction between the components is modeled as control constraints. The approach aims at controlling the components and also satisfying constraints specified using controlled languages. Indeed, the supervisor tries to respect the constraints by inhibiting some behaviors to provide the correct behavior of the originally uncontrolled DES.

Evaluation

The authors introduce a solution that is interesting from a theoretical point of view. They propose a polynomial-time algorithm that allows to respect the constraints and ensure a large set of correct and optimal behaviors of the originally uncontrolled systems that are modeled as finite state automata. However, this solution has some drawbacks. It is very difficult to implement and does not tackle the state space explosion problems. Therefore, it does not easily scale to real systems where the generated controller may have an enormous number of states even for small systems. For this, it has been extended by control theoretic ideas (*e.g.*, controllability, modular, and decentralized control) and several algorithms have been improved in numerous ways [136].

3.3.2 Modular Discrete Controller Synthesis Using Synchronous Language BZR

Other syntheses techniques are based on synchronous languages, such as BZR, that provide formal methods for the specification of reactive systems, feature verification tools, and code generation executable from the specification.

3.3.2.1 Synchronous Synthesis Techniques for Coordinating Adaptive and Reconfigurable Systems

In [55], Delaval *et Al.* present an extension of the Heptagon/BZR language, a reactive programming language, with behavioral contract constructs. This programming language is used to specify discrete control part of systems with automata. The authors introduce an approach based on contract enforcement and components abstraction to apply a modular discrete controller synthesis (DCS) on synchronous programs. Namely, each program is represented as a composition of nodes, which are the abstraction level used in BZR in order to achieve a modular application of the DCS. Each node is compiled in an independent way and has a contract with itself. Indeed, the DCS synthesizes a set of reconfiguration controllers for the program. It computes automatically, for each abstraction, a controller. Then, all the controllers are recomposed to generate a controller for the initial program. The generated controller must satisfy properties given as objective for the co-execution of the initial program. This approach stems from both control of discrete event system and control theory. The authors integrate it in a high-level programming language combining data-flow and automata machines, named BZR. This synchronous language defines reactive systems using generalized Moore machines. In turn, BZR has been used in several works (*e.g.,* component based software, the coordination of administration loops). This approach is applied to adaptive and reconfigurable systems, such as embedded systems. It was implemented in the Sigali tool [101], a symbolic model checker and synthesis controller that uses as input a global view of the system.

3.3.2.2 Synchronous Synthesis Techniques for Coordinating Autonomic Managers

In other recent works [54, 74, 75], Delaval, Rutten, Gueye *et Al.* propose some techniques based on synchronous discrete controller synthesis for coordinating autonomic managers, such as self-repair and self-sizing managers. The communication between the generated controller and managers is synchronous and uses the reactive and synchronous language BZR. BZR contains a behavioral contract mechanisms and dynamic temporal properties, which are enforced by an automatically build controller. The use of BZR is based on the notion of contract in system modeling. The contract is composed of three parts: (1) `enforce` is the part where the properties are declared as control objectives, (2) `with` is the part where some local controllable variables are added when the model that describes the system operation does not respect the properties, and (3) `assume` is the part that contains relevant properties of the run-time environment. This information is taken into account when synthesizing the control logic.

3.3.2.3 Evaluation

Compared to monolithic synthesis, this approach has the advantage of breaking down the synthesis computation cost. But, it also has some drawbacks which are:

— BZR cannot impose a specific order between requirements.

— this approach allows the designers to describe the constraints that must be ensured at runtime to compute the corresponding controller. However, the approach does not explain how the designers ensure that the constraints do not lead to an error-prone expression;

— the computed controller describes all the possible behaviors, which can be allowed in every step. The implementation of the controller needs a decision system that chooses only one of them;

— this approach does not provide additional information (*e.g.,* the false constraint on the contracts) to the designers when the controller synthesis fails;

— this approach does not propose solutions to deal with the synthesis failure.

3.3.3 Synthesis of Decentralised Supervisors for Distributed Adaptive Systems

A decentralized supervisory control approach for synchronous reactive systems is presented by Belhaj *et Al.* in [128, 127]. This work is based on distributed adaptive systems modeled through adaptation managers as synchronous reactive systems. The reactive systems, based on a set of events and states, are modeled as finite state machines, so-called Moore machines. Each machine contains a set of input/output signals and transition guards. The adaptation managers interact with the environment and continuously react to input signals by manipulating them and producing output signals. It is worth noting that the adaptation managers read simultaneously the values of input signals and compute instantly the output signals. This approach is based on existing techniques for discrete controller synthesis, to control the decentralized reactive systems. Each system is regarded as a composition of subsystems hosted on remote sites. Every local subsystem computes abstracted behavior models of the other distant subsystems by using observable variables, which it may observe. To do this, every local subsystem should benefit from the maximum of available information in order to reflect detailed abstract behaviors that are the most similar to the real subsystems. Then, discrete controller synthesis is used with abstracted behavior models to automatically generate local controllers on the remote sites (see Figure 3.7). Once generated, every local controller acts collectively and independently on its corresponding subsystem to ensure global safety properties, which may be decoupled into local properties.

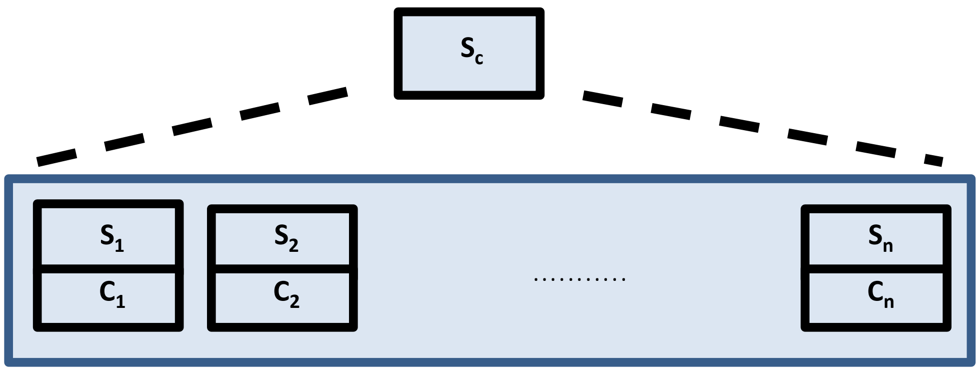


Figure 3.7 – Control of a decentralized reactive system, regarded as subsystems, through local controllers on each subsystem

Evaluation

The approach was applied to several examples for validation purposes and allows decentralized control consisting of several subsystems spread across remote sites. Each remote subsystem benefits of computing abstracted behavior models. Moreover, the local controllers rely on synchronous systems and synchronous communication semantics, whereas we assume asynchronous systems and communication, meaning that the controllability hypothesis is impossible in our context.

3.3.4 A Framework for the Coordination of Multiple Autonomic Managers in Cloud Environments

In [110], Oliveira *et Al.* present a framework for the coordination of multiple dependent autonomic managers in the cloud computing context. Managers are broadly classified in two categories: application-related managers that manage applications on the SaaS layer and infrastructure-related managers which manage the IaaS layer. The two categories use a protocol based on inter-manager events and actions and synchronization mechanisms for coordinating these managers. The suggested framework defines how the managers must communicate using these actions and events. The authors also propose some analytical models that are based on constraint programming and allow one to equip the managers with self-optimization capabilities.

Evaluation

This approach focuses on quality of service whereas our focus was on behavioral and functional aspects of the system execution.

3.3.5 Summary

We have presented a review of different controller synthesis techniques, which use languages and models used for coordination purposes (see Section 3.2). The coordination techniques use constraints, shared information, or event-driven coordination models and languages for coordinating entities. In our approach, we use constraints to describe the controller that we want to generate. We observe that most controller synthesis techniques are stemming from synchronous languages, whereas our approach assumes that communication is achieved asynchronously. Finally, we observe that not all these approaches use verification techniques to verify the control behavior.

Table 3.2 – Comparison between controller synthesis techniques

	Synchronous	**Asynchronous**	**Verification**	**Coordination**
Ramadge, Wonham	✓	✗	✓	constraints

Delaval *et Al.*	✓	✗	✓	contracts
Belhaj *et Al.*	✓	✗	✓	shared data
Oliveira *et Al.*	✗	✓	✗	constraints
Our techniques	✗	✓	✓	requirements

4

Dynamic Management Protocol for Cloud Applications

" There is a tremendous difference between a computing system that works and one that works well. "

K. Birman, R.V. Renesse, and W. Vogels [23]

Contents

We present in this chapter a novel protocol, which targets automatic deployment and (re)configuration of cloud applications. These applications are intricate distributed applications composed of virtual machines (VMs) running a set of interconnected software components. A reconfiguration manager guides the reconfiguration tasks by instantiating new VMs or destroying/repairing existing VMs. This reconfiguration manager may also execute architectural changes to the application by adding new components or removing existing components hosted on specific VMs. After the creation of new components due to the instantiation of a new VM or to components addition requests, the protocol is in charge of starting all components in the correct order according to the architectural dependencies. Each VM embeds a local reconfiguration agent (agent for short in the rest of this dissertation). Each agent interacts with the other remote agents. For each component, the VM agent tries to satisfy its required services by connecting them to their providers in order to finally start the component. The component cannot be started before the components on which it depends for mandatory imports. The supplier of the service can be hosted on the same VM or on another VM. When an agent receives a component removal or a VM destruction request from the reconfiguration manager, it tries to stop and unbind all the components hosted on the VM. A component cannot stop before all partner components connected to it have unbound themselves. To exchange messages and bind/start/unbind/stop components, the VMs communicate together through a publish-subscribe messaging system. The protocol is fault-tolerant in the sense that it enables the detection of VM failures that occur to a running application and repair them. When a VM failure occurs, the protocol notifies the VMs that are impacted. The protocol supports multiple failures. It always succeeds in finally reconfiguring the application at hand and stopping/starting all components. Our management protocol implies a high degree of parallelism. Hence, we decided to use formal methods and tools to specify, verify the protocol, and ensure that it preserves important architectural invariants (*e.g., a started component cannot be connected to a stopped component*) and satisfies certain properties (*e.g., each VM failure is followed by a new creation of that VM*). The protocol was specified using the specification language LOTOS NT (LNT) and verified with the CADP verification toolbox. For verification purposes, we used 600 hand-crafted examples (application models and reconfiguration scenarios). For each example, we generated the Labelled Transition System (LTS) from the LNT specification, and we checked on them about 40 properties that must be respected by the protocol during its application. To do so, we used the model checking tools available in the CADP toolbox. These formal techniques and tools helped us to improve the protocol by (i) identifying two important issues that we detected, and by then (ii) correcting them in the corresponding Java implementation which was written at the same time as the specification by another partner from the OpenCloudware project.

4.1 Reconfiguration Protocol

This section successively presents the application model, the protocol participants (*i.e.*, the reconfiguration manager, the publish-subscribe messaging system, and the reconfiguration agents), the protocol itself including the different possible reconfiguration operations, and an overview of the Java implementation.

4.1.1 Application Model

In this section, we first introduce an abstraction of the model, which is sufficient for describing the protocol principles. The real model exhibits more details, such as port numbers, URLs, and other implementation details. The model we use here is used for reconfiguring distributed cloud application. Its primary role is to keep track of the VMs and components currently deployed in a cloud application.

Distributed cloud applications are composed of a set of interconnected software components hosted on different virtual machines (VMs). Each one of these components is willing to provide services via exports and/or require services via imports. Ports are typed and matched when they share the same type. In fact, each import requires an export with the same type for being satisfied. An export can provide its service to several components and can thus be connected to several imports. An import should be connected to an

export, supplied by a component hosted on the same machine (local binding) or a component hosted on another machine (remote binding). Each import may be mandatory or optional. A mandatory import represents a service required by the component to be functional. Therefore, if the component needs mandatory imports, it cannot be started before all its imports are satisfied (*i.e.*, all mandatory imports are connected to started components). On the other side, an optional import is a service needed by the component, but not necessary for starting it. When a component has an optional service, it can be started before the component that provides it, which makes our model supports circular dependencies that involve at least one optional import. Hence, the connection of an optional import may be achieved after the start-up of the service requester and provider. Our protocol does not distinguish port types during the port resolution and (un)binding phases, but does when starting / stopping components.
The component used in this model has three states, namely, stopped, started, and failed. Initially, when the VM is instantiated, all its hosted components are stopped. For each component, once all its mandatory imports are bound to started components, the component can be started. Reversely the component must stop when at least one of the components to which it is connected through mandatory import requires stopping. When the VM fails, all its components are failed.

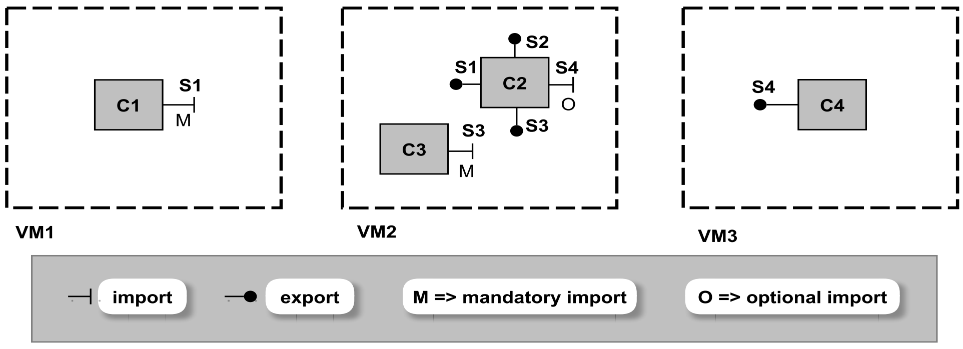


Figure 4.1 – Example of an application model (before binding ports)

We present in Figure 4.1 an example of an application model composed of three VMs. The first one (VM1) hosts one component C1 that has a mandatory import that is S1. Therefore, it cannot be started before the start-up of the S1 provider. The second VM (VM2) hosts two components, which are C2 and C3. C2 provides three services that are S1, S2, and S3. It has an optional import that is S4. Thus, it can be started immediately before the start-up of S4 provider. C3 needs a mandatory service S3. Therefore, it cannot be started before the start-up of this service provider. The third VM (VM3) hosts one component C4, which does not require any service. Therefore, it can be started immediately.

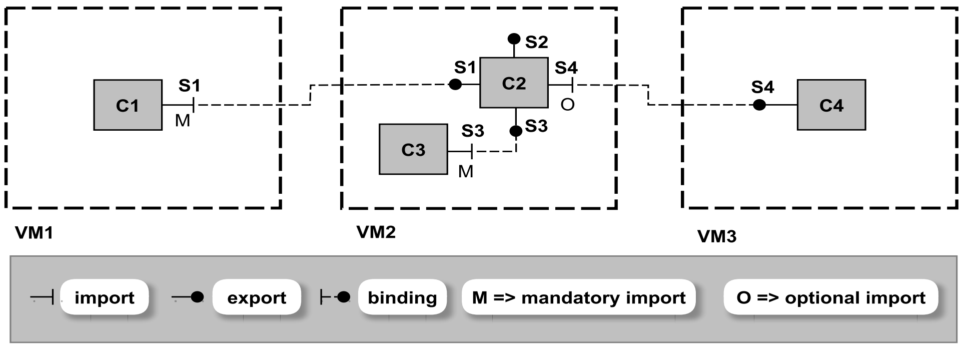


Figure 4.2 – Example of an application model (after binding ports)

Figure 4.2 shows how C1 is connected to C2 via the mandatory import, S1 (remote binding). Therefore, it cannot start before the start-up of C2. C2 needs only an optional import, named S4, and provided by C4 and thus, it can be started immediately. However, it cannot be connected to C4 before the start-up of this last one, which prevents to have a started component bound to a stopped one. C3 is connected to C2 via a mandatory import (local binding). Therefore, it cannot start before the start-up of C2. Finally, C4 does not require any service. Therefore, it can be started immediately.

4.1.2 Architectural Invariants

As stated before, the architectural invariants must hold for any reconfiguration of distributed cloud applications. We present below the set of invariants that we defined to preserve the application consistency.

— there are no circular dependencies through mandatory imports;
— a component cannot be started before the component it depends on for mandatory imports;
— a started component cannot be connected to a stopped component;
— there is no component connected to failed or removed components.

The satisfaction of these invariants during the application execution is an important goal in the context of cloud applications.

4.1.3 Protocol Participants

The management protocol consists of three following participants: the reconfiguration manager (RM), virtual machines (VMs), and the publish-subscribe communication messaging system (PS), depicted in Figure 4.3.

4.1.3.1 Reconfiguration Manager (RM)

The RM is responsible for reconfiguring the application. It is used to perform two main roles: (1) it guides the application reconfiguration by successively sending reconfiguration operations, which are described in a reconfiguration scenario and indicate changes to perform on the application. Namely, the reconfiguration operations are instantiating a new VM, destroying an existing VM, repaired a failed VM, adding a new component, and removing an existing component. (2) The RM is also in charge of repairing each VM failure by creating a new instance of the failed VM when a failure is detected.

4.1.3.2 Virtual Machine (VM)

Each VM is associated with a reconfiguration agent (agent for short in the rest of this dissertation). Each agent interacts with the other participants and ef-

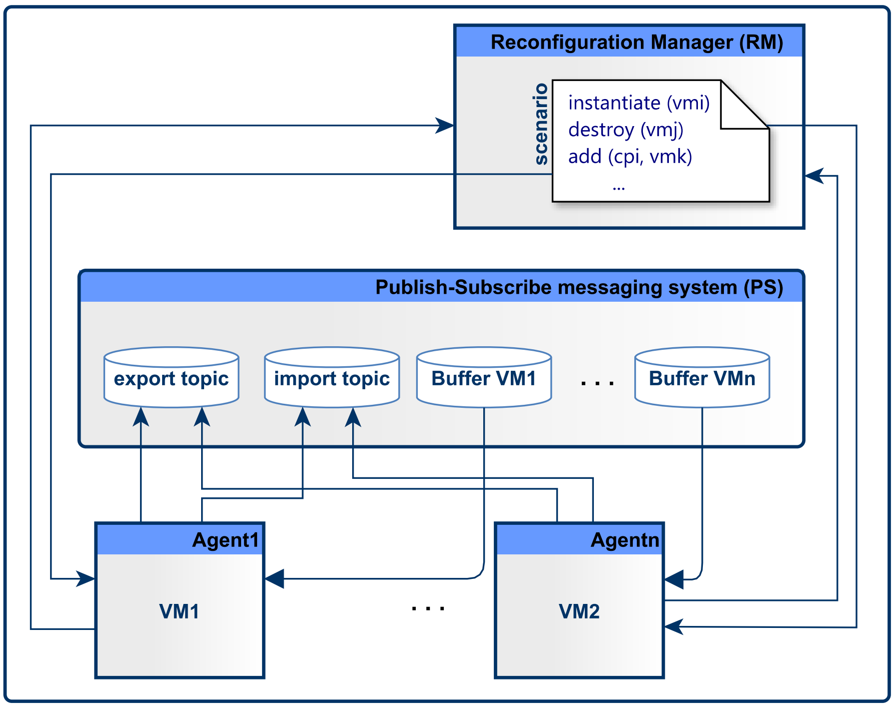


Figure 4.3 – Protocol participants: reconfiguration manager, VMs, and publish-subscribe messaging system

fectively manages its own VM. It has certain goals to achieve, such as connecting bindings and starting components in response to the VM instantiation or the adding reconfiguration operations received from the RM. The agent should also disconnect bindings and stop components upon reception of a destruction VM or component removal operations from the RM.

4.1.3.3 Publish-Subscribe messaging system (PS)

The PS supports the exchange of communications among all VMs. When it receives a message containing some information (*e.g.*, binding details, component state) from a VM, it transfers it to the VMs that need that information. In order to ensure that this is done, the PS is equipped with a set of FIFO buffers (a buffer for each VM). A buffer for each new VM is added to the PS once that VM is instantiated to the application. Once an existing VM is destroyed from the application, its buffer is immediately removed from the PS. Each buffer is basically used to store the messages that are sent to its VM and that can contain useful information for the start-up or shutdown. We assume that transferred messages are never lost during this communication. The PS is also equipped with two topics[1]: *(i)* an export topic where a component subscribes its imports and publishes its exports, and *(ii)* an import topic where a component subscribes its exports and publishes its imports. We show in Section 4.1.4.1 why this double subscription/publication is required.

4.1.4 Protocol Description

We explain in this section how the protocol works in order to perform its main goal, which is to reconfigure dynamically distributed cloud applications by instantiating new VMs/destroying existing VMs, adding a new component to an existing VM/removing a component from its hosting VM, and repairing failures. We detail how it works in each of these situations.

4.1.4.1 Start-up

The first goal of an agent is to start all the local components when instantiating a new VM or when adding a component to an existing VM. The agent behavior, illustrated in Figure 4.4, aims to start each component having mandatory and/or optional imports. First, when the agent receives an add component request from the RM (❶), it checks the presence of exports and import for that component. If the component provides services, it subscribes to the import topic and then publishes each of these services to the export topic (❷). Then, the agent reacts following the list of imports.

No import

If the component does not have any import, it starts immediately (❼) and then, sends a started message to the PS (❽) to notify it about its new state that

1. A topic is a logical channel where messages are published and subscribers to a topic receive messages published on that topic.

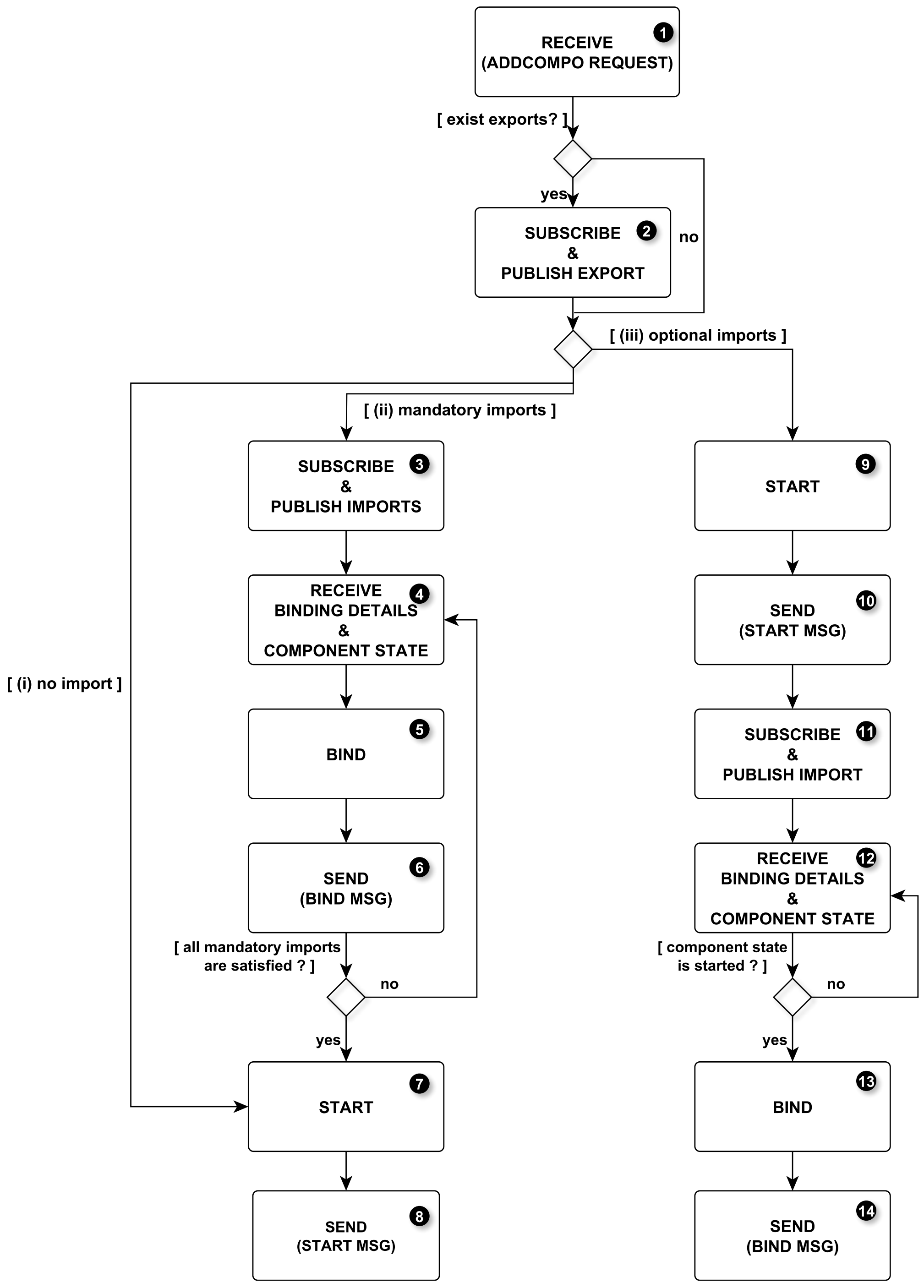

RECEIVE (ADDCOMPO REQUEST) 1
[exist exports?]
yes
no
SUBSCRIBE & PUBLISH EXPORT 2
[(iii) optional imports]
[(ii) mandatory imports]
[(i) no import]
SUBSCRIBE & PUBLISH IMPORTS 3
RECEIVE BINDING DETAILS & COMPONENT STATE 4
BIND 5
SEND (BIND MSG) 6
[all mandatory imports are satisfied ?]
no
yes
START 7
SEND (START MSG) 8
START 9
SEND (START MSG) 10
SUBSCRIBE & PUBLISH IMPORT 11
RECEIVE BINDING DETAILS & COMPONENT STATE 12
[component state is started ?]
no
yes
BIND 13
SEND (BIND MSG) 14

Figure 4.4 – The agent behaviour when adding a new component: (i) without imports; (ii) with, at least, a mandatory import; and (iii) with, only optional imports

is started.

Mandatory import

To be functional, the component expects that all its mandatory imports will be connected to exports with the same types provided by started components. The service provider can be a component hosted on the same VM (local) or on another VM (remote). For each mandatory import, the component subscribes to the export topic and then publishes the import to the import topic (❸). When the PS receives the publication message, it checks the import topic. If it does not find any provider for the service, that message is deleted. Otherwise, if the PS finds more than one service provider, it picks one randomly. Then, it sends a message containing binding details and provider state to the component. The import can connect to the service provider (❺) upon reception of that message from the PS (❹). Finally, the agent sends a binding message (❻) to the PS informing it about the connection to the service provider. Every time the component receives a message containing binding details/and or component states, it verifies the provider state and if all mandatory imports are connected. If it finds that all its mandatory imports are connected to started components, the VM agent starts the component (❼) and then, sends a started message to the PS notifying it about its new state that is started (❽).

Optional import

When the component has only optional imports, it can immediately start even if these imports are not satisfied (❾) and send a started message to the PS (❿). Then, for each optional import, it subscribes to the export topic and publishes the import to the import topic (⓫). Upon reception of a message with binding details and component state (⓬), the import binds (⓭) to this compatible export and then sends a binding message (⓮) only if the component state is started.

When the component has several optional imports and at least one mandatory import, it cannot start before satisfying the mandatory import (bind it to the service provider and the partner component has been started).

When the RM guides the application reconfiguration by instantiating a new VM, a buffer for this VM is added to the list of buffers contained in the PS.

The agent tries to start all components hosted on the added VM. For this, it considers each component hosted on the new VM like a new component that will be added to an existing VM.

Importance of the double subscription/publication

To illustrate how the PS is used for resolving port dependencies and start/stop components, we focus on two concrete scenarios for deploying a part of the application presented in Figure 4.1. This part includes only VM1 and VM2, where VM1 contains C1 and VM2 contains only C2. In the first scenario, we instantiate VM1 and then VM2 (see Fig. 4.5), whereas they are instantiated in the other way round in the second scenario (see Fig. 4.6).

1. **First scenario: Instantiating VM1 then Instantiating VM2** When VM1 is instantiated, C1 requires a mandatory import and does not provide any service. Therefore, it subscribes to the export topic (A1.1) and then publishes its import to the import topic (A1.2). The PS receives the publication message, checks the import topic, but it does not find a provider of this required service (there is no component subscribed to the import topic with the S1). The publication message is deleted. When VM2 is instantiated, C2 requires an optional import and can, therefore, be started immediately (A2.1). Its state changes to started. Then, it subscribes to the export topic and publish its import to the import topic. C2 provides three services with types S1, S2, and S3. Therefore, for each export, it subscribes it to the import topic (A2.2) and then publishes it with its state to the export topic (A2.3) (we show only the S1 service for simplifying the Figure 4.5). The PS receives the publication message corresponding to S1 from the VM2 agent, checks the export topic and finds that C1 hosted on VM1 has required this service (C1 has subscribed to the export topic). Hence, a message with binding details and C2 state is added to VM1 buffer (PS.1). Upon reception of this message, the C1 component is bound to the C2 component (A1.3). The state of C2 component is started (8). Therefore, VM1 agent starts the C1 component (A1.4). The application is fully operational.

2. **Second scenario: Instantiating VM2 then Instantiating VM1** When

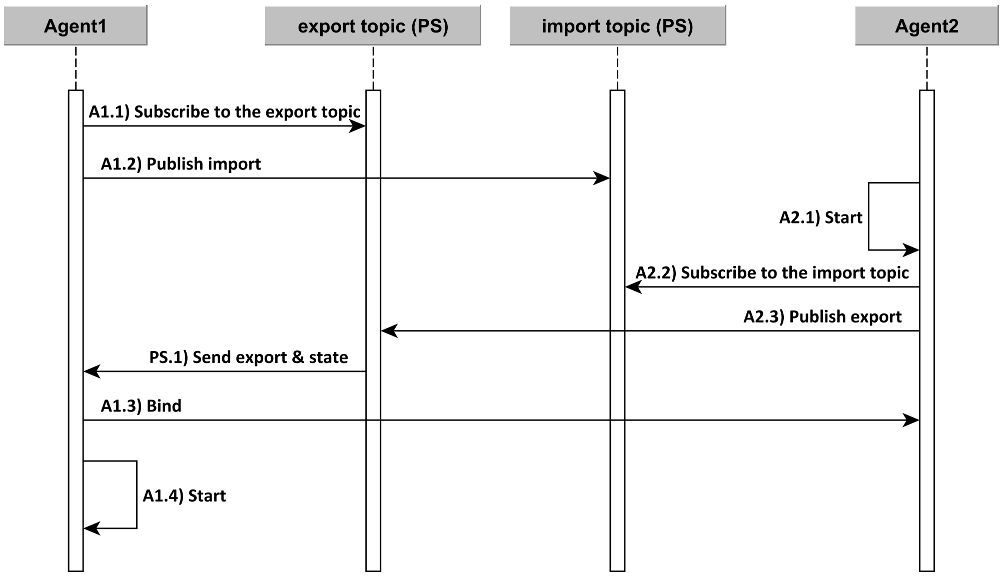


Figure 4.5 – Instantiating VM1 then Instantiating VM2

VM2 is instantiated, the C2 component has only an optional import. It is, therefore, started immediately (A2.1) and its state changes to started. C2 provides three exports with types S1, S2, and S3. So, for each export, it subscribes it to the import topic (A2.2) and then publishes it with its new state (started) to the export topic (A2.3). The PS receives the publication message corresponding to S1 from the VM2 agent, checks and does not find any component that requires it. Therefore, the publication message is deleted. When VM1 is instantiated, C1 component requires a mandatory service whose type is S1. Therefore, it subscribes to the export topic (A1.1) and then publishes its import to the import topic (A1.2). When PS receives the publication message from the VM1 agent, it checks the import topic and finds that C2 has provided the S1 service (it has subscribed to the import topic). So, it notifies VM2 that here is a component (C1) hosted on VM1 that needs S1 (PS.1). VM2 receives the notification message, so it publishes C2 export and state, which is started (A2.4). The PS forwards this message to the VM1 agent (PS.2), and the C1 component can finally be bound to the C2 component (A1.3) and started (A1.4).

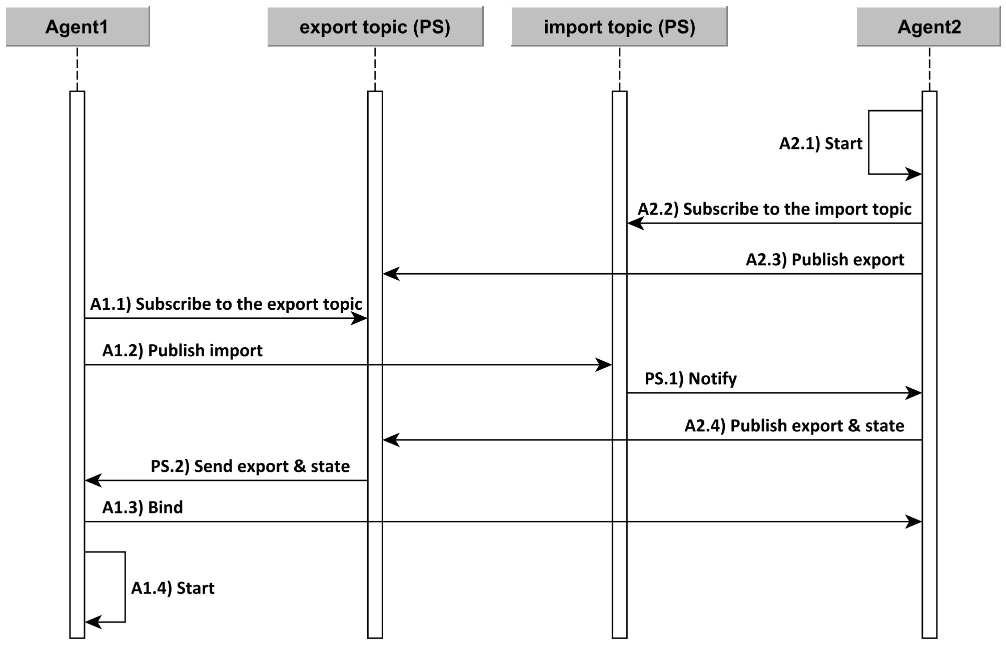


Figure 4.6 – Instantiating VM2 then Instantiating VM1

4.1.4.2 Shutdown

We describe in this section the second goal of the management protocol which consists in removing existing components from a VM or destroying an existing VM from a running application.

Removing a component

When the RM requires the removal of a component from its VM, it sends a remove component request to the agent of the VM that contains the component. Upon reception of the request, the agent tries to stop the component and then remove it without violating any architectural invariants (*e.g.*, a started component cannot be connected to a stopped component). Most importantly, the way to stop a component depends on its list of exports (see Fig. 4.7). We distinguish the two following cases:

1. **Empty exports list.** When the component does not provide any service, it stops immediately (❺). Then, it unbinds all its connections (❻) and sends an *"unbind confirmed"* message for each import (❼).

2. **Export list with at least one export.** When the component provides services, its VM agent cannot stop it before that all components bound to it through mandatory imports stop and then unbind in order to restore consistency of the whole application. Components bound to it through optional imports, need only to unbind without stopping. To do so, for each export, it unsubscribes from the import topic (❷) and then sends a message *"ask to unbind"* to the PS (❸). Then, it waits until all components bound to it through mandatory imports disconnect. Components bound to it through optional imports need only to disconnect. When the PS receives an *"ask to unbind"* message, it transmits it to all components subscribed to that export. The PS notifies the component that need to stop each time a component connected to it has unbound by sending an *"unbind confirmed"* message (❹). Once all components, connected to it, have effectively unbound, it can finally stop itself (❺), unbind all its imports (❻), and send an *"unbind confirmed"* message to the PS (❼).

Asking to unbind a component

When a component receives an *"ask to unbind"* message (❶) (see Fig. 4.8), it can remain in the same state and then unbinds directly if it is bound through an optional import (❺). If it is bound through a mandatory import, the protocol handles it as done to the component that needs to be removed. The only difference between stopping a component that will be removed and a component that receives an *"ask to unbind"* message is that, the second one should not unsubscribe from the import topic if it provides exports. Moreover, after unbinding, it publishes its import in order to connect to another export proposed by the publish-subscribe system (❼).

The component shutdown implies a backward propagation of *"ask to unbind"* messages and, when this first propagation ends (on components with no export or optional imports only), a second forward propagation of *"unbind con-*

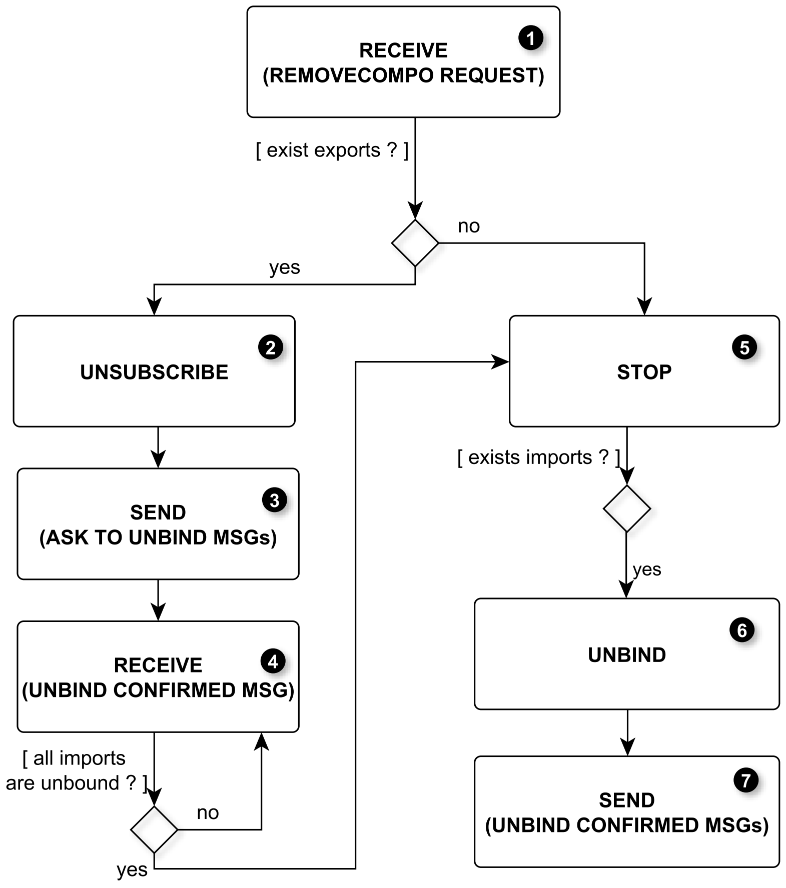

RECEIVE (REMOVECOMPO REQUEST) 1
[exist exports ?]
no
yes
UNSUBSCRIBE 2
SEND (ASK TO UNBIND MSGs) 3
RECEIVE (UNBIND CONFIRMED MSG) 4
[all imports are unbound ?]
no
yes
STOP 5
[exists imports ?]
yes
UNBIND 6
SEND (UNBIND CONFIRMED MSGs) 7

Figure 4.7 – The agent behavior when removing a component

firmed" messages starts to let the components know that the disconnection has been actually achieved.

When the RM requires destroying a VM, it sends a destruction VM request to the VM agent. This agent must properly stop all the components hosted on that VM as well as all components bound to them. Therefore, for each component hosted on the VM that will be destroyed, it behaves as if it has received a remove component request. Once all components are removed, the VM is finally destroyed.

4.1.4.3 VM Failures

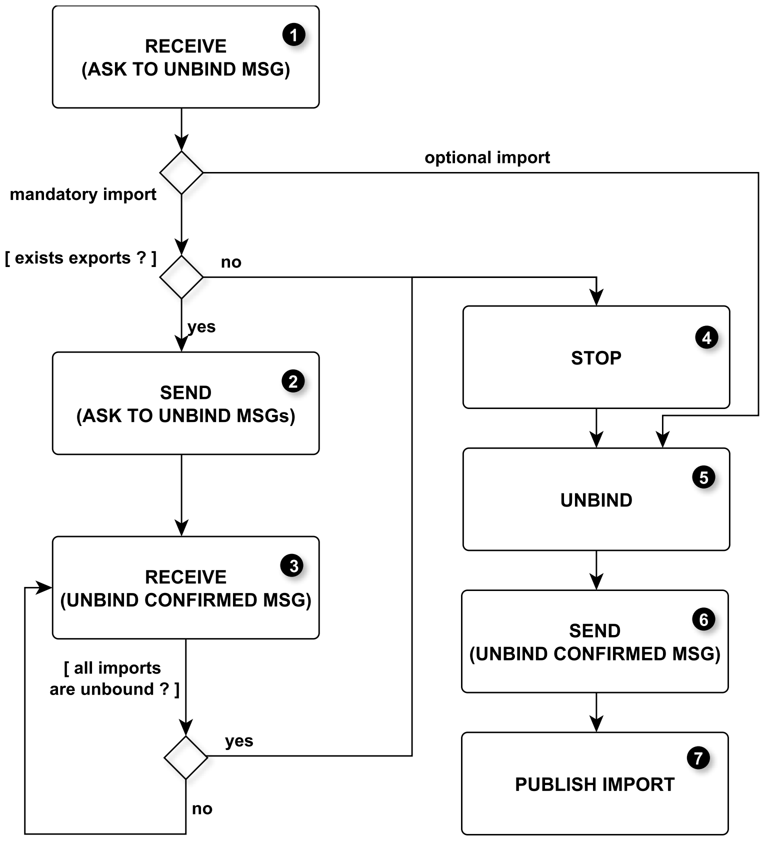

RECEIVE (ASK TO UNBIND MSG) 1
optional import
mandatory import
[exists exports ?]
no
yes
SEND (ASK TO UNBIND MSGs) 2
RECEIVE (UNBIND CONFIRMED MSG) 3
[all imports are unbound ?]
yes
no
STOP 4
UNBIND 5
SEND (UNBIND CONFIRMED MSG) 6
PUBLISH IMPORT 7

Figure 4.8 – The agent behavior when stopping a component

VM failures and component failures can occur at any time to a running application (*e.g.*, during a reconfiguration). We present in this section only the VM failure. A component failure is a particular case of a VM failure. When a VM fails, all its hosted components stop unexpectedly without alerting any partner components. Therefore, we can be in a situation where started components are connected to failed components. In order to restore the application consistency (*i.e.*, architectural invariants), VM failures must be handled immediately. We explain in this section how the protocol takes into account VM failures and how it works to manage them (Fig. 4.9). The protocol also supports the case of multiple failures.

When a VM is instantiated, its agent interacts with the RM by periodically sending an *"alive"* message. The RM is configured to receive this message periodically from each agent. As long as the RM receives these *"alive"* messages from each agent within a specific delay, it considers that its VM is working correctly and still alive [22] (❶). Once the RM does not receive one of these *"alive"* messages from an agent before the delay elapses, it assumes that its VM has failed (❷). In that case, it alerts the PS (❸) and then starts the repair phase by creating a new instance of the failed VM (❹). The new VM behaves as if the machine is instantiated for the first time (see Section 4.1.4.1).

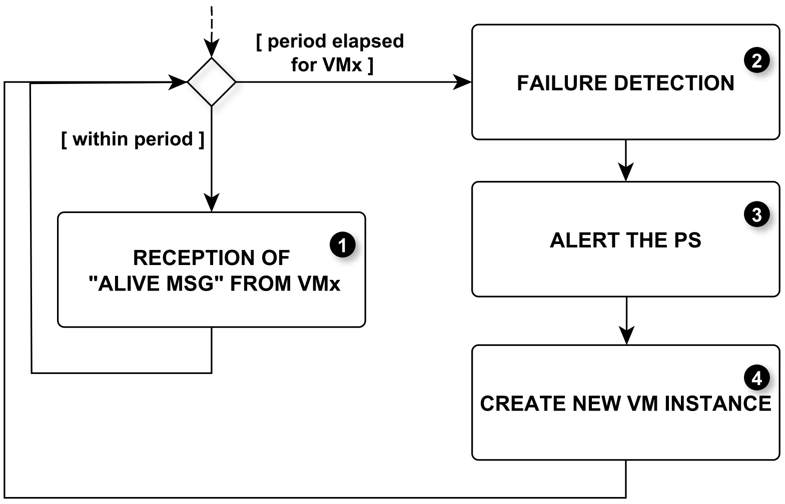


Figure 4.9 – The RM behavior when detecting a VM failure

Figure 4.10 shows that when the PS receives an *"alert failure"* message from the RM (❺), it unsubscribes each failed component exporting services from the import topic to avoid that other components try to connect to it (❻). Then, it warns the other VMs of this failure by:

— sending messages notifying all components bound to the failed components about the VM failure and asking them to unbind (❼). When an agent receives this kind of message, it stops immediately the impacted component if it is bound through a mandatory import without waiting the disconnection of components bound to it. The impacted component remains in the same state if it is bound through an optional import. Then, it unbinds it and tries to connect it to another import provider in order to restart the component as presented in Section 4.1.4.1;

— sending an *"unbind confirmed"* message to each component to which a failed component was connected via mandatory or optional import (❽). The reception of this message by a component that needs to stop indicates the disconnection has been actually achieved.

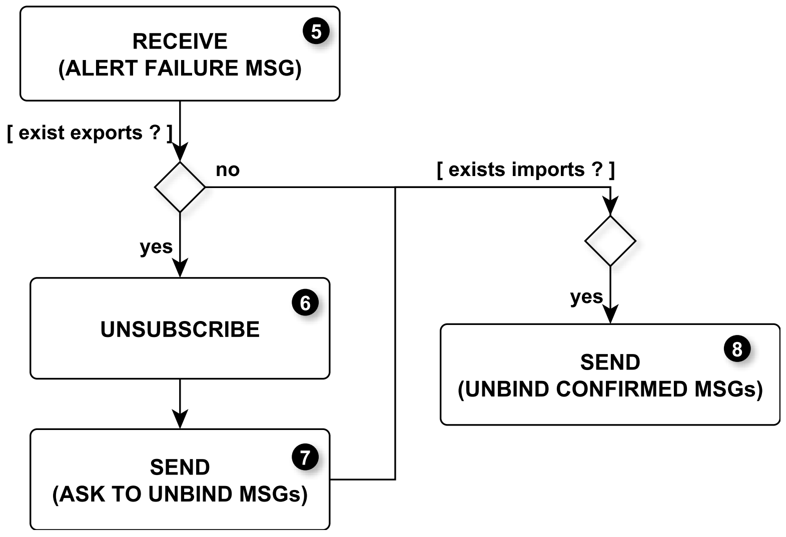

RECEIVE (ALERT FAILURE MSG)
5
[exist exports ?]
no
[exists imports ?]
yes
UNSUBSCRIBE
6
yes
SEND (UNBIND CONFIRMED MSGs)
8
SEND (ASK TO UNBIND MSGs)
7

Figure 4.10 – The PS behavior when receiving an alert failure message

When a component fails, the agent of the VM on which it is hosted informs the PS and then creates a new instance of the failed component. The new component behaves like a component added to the application for the first time (see Section 4.1.4.1). When the PS receives the component failure alert, it reacts in the same way as for each component hosted on a failed VM.

4.1.5 Example of a Component Removal/Addition and VM Failure

This section successively introduces an example of removal and addition of a component to the application showed in Figure 4.11, as well as an example of VM failure. In this example, we begin with a running application where all components are connected and started. The application is composed of three VMs. The first one (VM1) hosts two components Apache1 and Apache2. Apache1 is bound to Tomcat, which is hosted on the second VM (VM2), through the mandatory import Workers1. Apache2 is bound to the same Tomcat server through the optional import Workers2. Finally, Tomcat is bound to a MySQL database named MySQL and hosted on the third VM (VM3) through the mandatory import data.

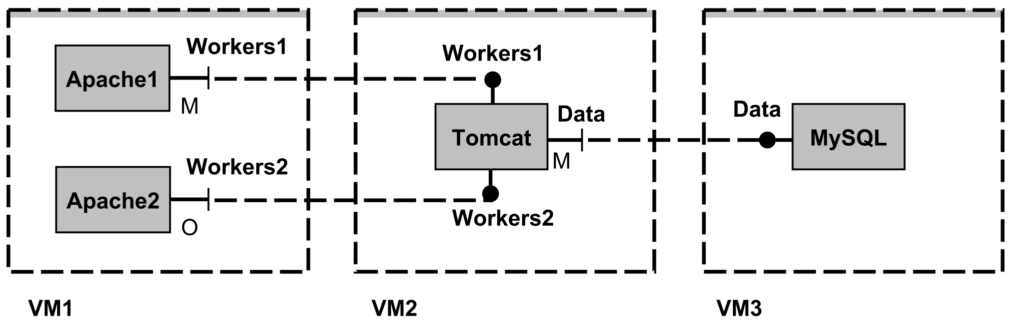


Figure 4.11 – Example of a running application composed of connected and started components

4.1.5.1 A Scenario of Component Removal

As it is showed in Figure 4.12, the RM requests the VM2 agent to remove Tomcat (RM.1). Once VM2 receives the request, it aims to stop the Tomcat component before removing it. To do so, all components connected to the Tomcat must be stopped before. Tomcat provides two services Workers1 and Workers2. Thus, for each export, it unsubscribes from the import topic (A2.1 and A2.3) and sends messages to the PS asking it to unbind all components connected to it through Workers1 and Workers2 (A2.2 and A2.4). The PS receives these messages, checks the export topic and finds that Apache1 (Apache2 respectively) hosted on VM1 imports Workers1 (Workers2 respectively) from Tomcat. Therefore, it sends *"ask to unbind"* messages to Apache1 and Apache2 (PS.1 and PS.2). When VM1 receives these messages, Apache1 does not provide any service and it is bound to the Tomcat component through a mandatory import, so it is immediately stopped (A1.1). Then, it is unbound from Tomcat, sends an *"unbind confirmed"* message to the PS (A1.2), and publishes its import to the import topic (A1.3). VM2 receives that message from the PS (PS.3) but cannot stop the Tomcat component because Apache2 is still connected to it. Apache2 is connected to the Tomcat component through an optional import. Thus, it is only unbound from Tomcat without stopping, then sends an *"unbind confirmed"* message to the PS (A1.4), and publishes its import to the import topic (A1.5). The PS checks the import topic, but there is no component that provides Workers1 or Workers2. VM2 receives the *"unbind confirmed"* message from the PS (PS.4). Tomcat has no component bound to it any more, so it is stopped (A2.5). Tomcat is unbound from MySQL, and the VM2 agent sends an *"unbind confirmed"* message to the PS (A2.6). The PS finally sends that message to the MySQL component (PS.5).

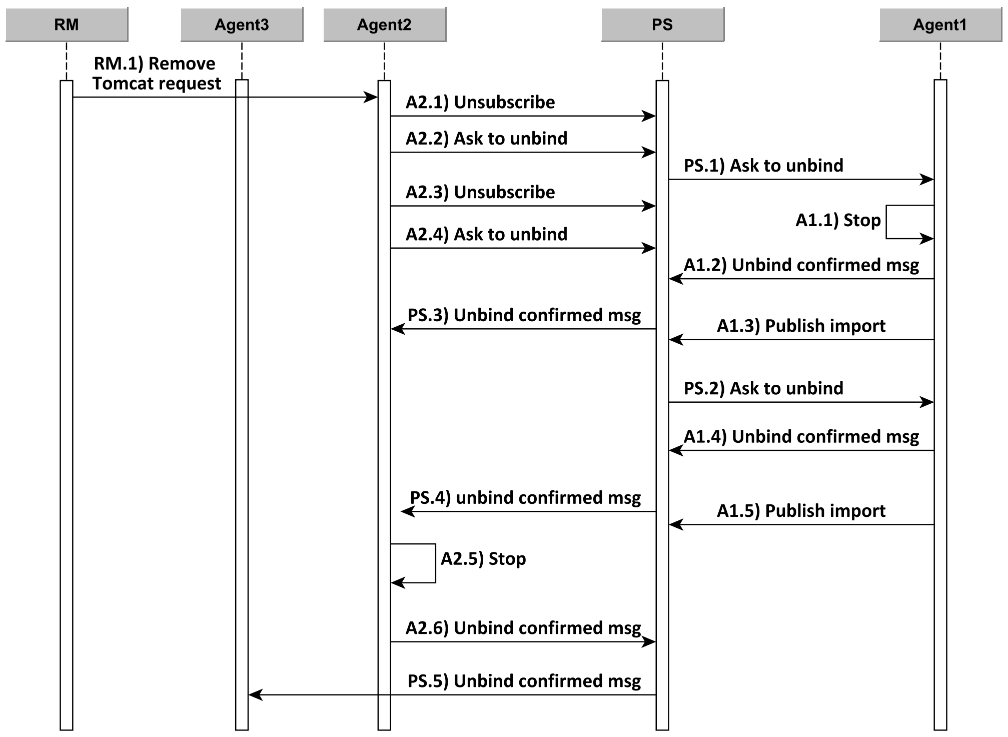


Figure 4.12 – The participants behavior when removing the Tomcat component

4.1.5.2 A scenario of component addition

In this section, an example of component addition to an existing VM is presented. We show in Fig. 4.13 how to add a new instance of Tomcat to VM2 after removing it in Section 4.1.5.1.

VM2 receives the Tomcat addition request from the RM (1). The Tomcat component requires a mandatory service whose type is Data. Therefore, it subscribes to the export topic and then publishes its import to the import topic (A2.1). Tomcat provides two services Workers1 and Workers2. Therefore, for each export it subscribes to the import topic and then publishes it to the export topic (A2.2 and A2.3). The PS receives the first message from the VM2 agent, checks the import topic and finds that only MySQL provides a Data service (it has subscribed to the import topic). The PS notifies VM3 that there is a Tomcat that needs Data (PS.1). When Tomcat publishes its exports, the PS forwards the binding details and Tomcat state to Apache1 because it has subscribed to the export topic for Workers1 (PS.2) (Apache2 respectively because it has subscribed to the export topic for Workers2 (PS.3)). When VM1 receives the binding details and Tomcat state, Apache1 connects to Tomcat (A1.1) but it cannot start because the Tomcat state is stopped. Apache2 cannot connect to Tomcat because it is started and Tomcat is still stopped. After receiving the notification message from the PS about the Tomcat component need (PS.1), VM3 sends the MySQL binding details and state that it is started (A3.1). The PS receives the start-up information from the VM3 agent, checks and finds that the Tomcat component has required this service (it has subscribed to the export topic). Hence, a message with binding details and MySQL's state is added to VM2 buffer (PS.4). Upon reception of this message, the Tomcat component is bound to the mySQL component (A2.4) and the VM2 agent starts the Tomcat component (A2.5). Then, the Tomcat component publishes a started message containing its new state (A2.6). The PS receives that message and forwards it to VM1 (PS.5). Upon reception of this message, the VM1 agent starts the Apache1 component (A1.2). Apache2 can finally connect to the Tomcat component (A1.3).

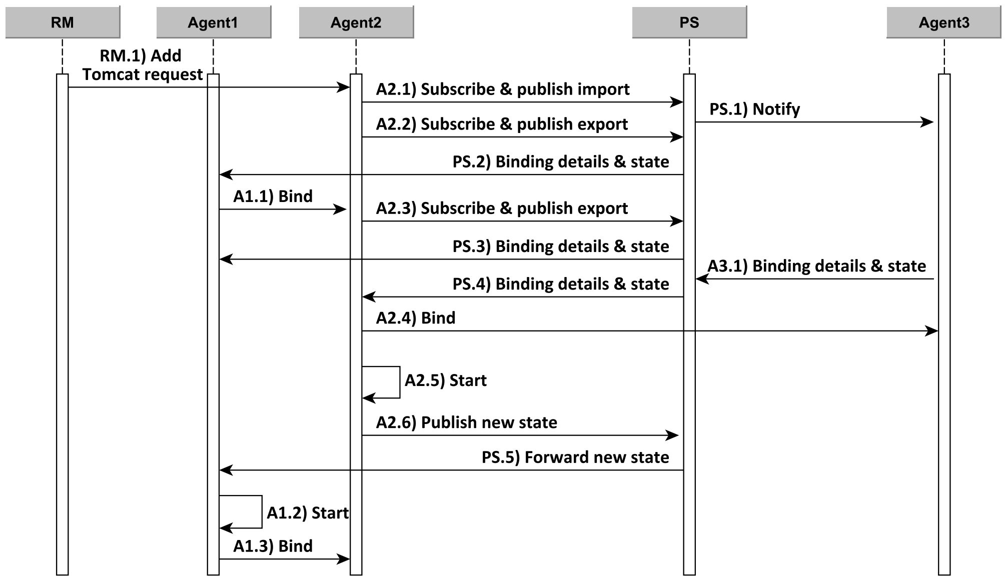

RM
Agent1
Agent2
PS
Agent3
RM.1) Add Tomcat request
A2.1) Subscribe & publish import
PS.1) Notify
A2.2) Subscribe & publish export
PS.2) Binding details & state
A1.1) Bind
A2.3) Subscribe & publish export
PS.3) Binding details & state
A3.1) Binding details & state
PS.4) Binding details & state
A2.4) Bind
A2.5) Start
A2.6) Publish new state
PS.5) Forward new state
A1.2) Start
A1.3) Bind

Figure 4.13 – The participants behavior when adding the Tomcat component

4.1.5.3 A scenario of VM Failure repair

We present in the remainder of this section an example of failure/repair of VM2 (see Fig. 4.14). When VM2 fails, Tomcat is suddenly stopped without alerting the components bound to it. Therefore, Apache1 and Apache2 that are started are bound to a stopped component. When the RM detects the failure of VM2 (RM.1), it immediately alerts the PS (RM.2). Then, the RM creates a new instance of VM2 (RM.3).
When the PS receives the alert message from the RM announcing the VM2 failure, it unsubscribes Tomcat from the import topic (PS.1). Then, it checks the export topic to find the impacted components. Thus, it finds that Apache1 and Apache2 are connected to Tomcat. Therefore, it notifies them about the failure (PS.2 and PS.3). When VM1 receives the notification messages, Apache1 does not provide any service and it is connected to Tomcat through a mandatory import, so it is immediately stopped (A1.1). It is also unbound from the Tomcat component, sends an *"unbind confirmed"* message to the PS (A1.2), and then publishes its import to the import topic (A1.3). Apache2 does not provide any service and is connected to the failed component through an optional import so it is unbound from Tomcat without stopping, sends an *"unbind confirmed"* mes-

sage to the PS (A1.4), and then publishes its import to the import topic (A1.5). After the creation of a new instance of VM2, each agent starts the components impacted by the failure as presented in Section 4.1.5.2.

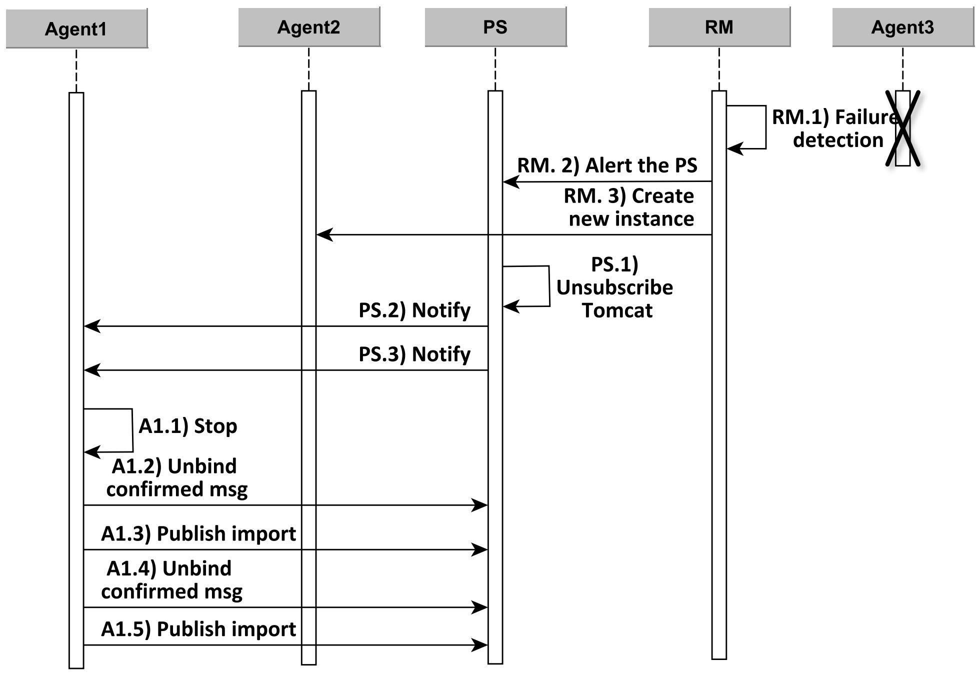


Figure 4.14 – The participants behavior when detecting the VM3 failure

4.2 Protocol Specification and Verification

Our management protocol implies a high degree of parallelism. Hence, we decided to use formal techniques and tools to specify and verify it. We ensure that it preserves important architectural invariants (*e.g.*, a started component cannot be connected to a stopped component) and satisfies certain properties (*e.g.*, each VM failure is followed by a new creation of that VM). The protocol was specified using the specification language LOTOS NT (LNT) [42], which is an improved version of LOTOS [80], and verified with the CADP verification toolbox [64]. For verification purposes, we used 600 hand-crafted examples (application models and reconfiguration scenarios). For each example, we generated the Labelled Transition System (LTS) from the LNT specification. We checked on them about 40 properties that must be respected by the protocol during its application. To do so, we used the model checking tools available in the CADP toolbox. These formal techniques and tools helped us improve the protocol by *(i)* detecting several issues and bugs, and by *(ii)* correcting them systematically in the specification and also in the corresponding Java implementation.

4.2.1 Protocol Specification

Thr LNT processes are built from actions, choices (`select`), parallel composition (`par`), looping behaviors (`loop`), conditions (`if`), and sequential composition (`;`). The communication between the protocol participants is carried out by rendezvous on a list of synchronized actions included in the parallel composition (`par`). The number of lines of processes depends on the size of the application model (the number of VM, component and ports), which represents the input of the protocol (see Fig. 4.15). Processes are generated for each input application model[2], because a part of the LNT code depends on the number of VMs and on their identifiers. Therefore, the number of lines for processes grows with the number of VMs in the application model. The specification of the protocol is described in the "reconfig.lnt" file and consists of three parts: data types (200 lines), functions (800 lines), and processes (1,500 lines). The number given above corresponds to an example with three VMs.

4.2.1.1 Data types

They are used to describe the distributed cloud application model (VMs, components, ports), the communication model (binding between components,

2. We developed an LNT code generator in Python, named reconf.py, for automating this task.

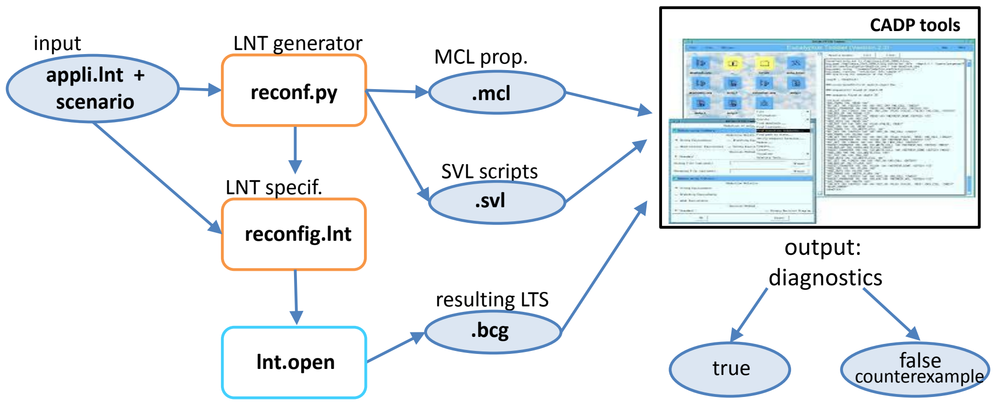


Figure 4.15 – Tool support

messages, buffers, and topics), and the component states. We present below a few examples of data types. The application model (`TModel`) consists of a set of virtual machines (`TVM`). Each VM has an identifier (`TID`) and a set of components (`TSoftware`). Each component (`TComponent`) has an identifier (`TID`), a set of imports (`TImport`), a set of exports (`TExport`), and a state (`TState`).

```
type TModel is
      set of TVM
end type

type TVM is
      tvm (idvm: TID, cs: TSoftware)
end type

type TSoftware is
      set of TComponent
end type

type TComponent is
      tcomponent (idcomp: TID, imports: TImport,
                        exports: TExport, state: TState)
end type
```

4.2.1.2 Functions

They apply on data expressions. Functions are used to define all the computations necessary for reconfiguration purposes (*e.g.*, extracting information from the application, describing buffers and basic operations on them like adding/retrieving messages, changing the state of a component, keeping track of the started/unbound components, verifying the satisfaction of the imports, etc.). Let us show, for illustration purposes, two functions that aim at removing (adding, resp.) a message from (to, resp.) a buffer by using the FIFO strategy. This strategy consists in removing the message from the beginning and adding a new message at the end of the buffer.

— The remove function takes as input a buffer (`q`) whose type is (`TBuffer`) that is composed of an identifier (`TID`) and a list of messages (`TMessage`). If the buffer is empty, nothing happens. When the buffer is not empty, the first message is removed.

```
function remove_MSG (q: TBUFFER): TBUFFER is
   case q in
      var name: TID, hd: TMessage, tl: TQueue in
         tbuffer(name,nil)
            -> return q
         | tbuffer(name,cons(hd,tl))
            -> return tbuffer(name,tl)
   end case
end function
```

— The add function takes as input the buffer (`q`) and the message that will be added. If the buffer is empty, the message is added. When the buffer is not empty, the function calls another function that adds the message to the end of the buffer tail, whose type is (`TQueue`).

```
function add_MSG (m: TMessage, q: TBUFFER): TBUFFER is
  case q in
    var name: TID, hd: TMessage, tl: TQueue in
      tbuffer(name,nil)
       -> return tbuffer(name,cons(m,nil))
     | tbuffer(name,cons(hd,tl))
       -> return tbuffer(name,cons(hd,add_queue(m,tl)))
  end case
end function

function add_queue (m: TMessage, tl_q: TQueue): TQueue is
  case tl_q in
    var hd: TMessage, tl: TQueue in
      nil           -> return cons(m,nil)
      | cons(hd,tl) -> return cons(hd,add_queue(m,tl))
  end case
end function
```

4.2.1.3 Processes

They are used to specify the participants of the protocol (the reconfiguration manager, the publish-subscribe messaging server, and an agent per VM). The reconfiguration manager guides the application reconfiguration. Each agent drives the behavior of its VM and encodes most of the protocol functionality in order to start/stop all the components hosted on its VM. The publish-subscribe messaging system assures the communication between all VMs. It is equipped with a set of FIFO buffers (a buffer for each VM in the application). Each participant is specified as an LNT process and involves two sorts of actions:

— actions which correspond to interactions with the other participants such as `PStoAGENT` that presents the message transferred from the publish-subscribe messaging system processes to the agent processes, `AGENTtoPS` that presents the message transferred from the agent processes to the publish-subscribe messaging system;

— actions which tag specific moments of the protocol execution such as the VM instantiation/destruction, the component start-up/shutdown, the component addition/removal, the effective binding/unbinding of an import to an export, the failure of a VM, etc.

For illustration purposes, we present the LNT process main (named `MAIN`) generated for an example of application model involving three VMs. The LNT parallel composition is described with `par` followed by a set of synchronization actions that must synchronize together. Two processes synchronize if they share the same action. We can see that the agents do not interact directly together and evolve independently from one another. The VM agents communicate together through the PS. Each agent is identified using the VM name and synchronizes with the PS on `AGENTtoPSi` action when sending a message to the PS and `PStoAGENTi` action (i = 1, 2, 3) when receiving a message from it. Each agent defines actions for port binding (`BINDCOMPO`), for starting a component (`STARTCOMPO`), for stopping a component (`STOPCOMPO`), etc. The PS is initialized with a buffer per VM and two topics for imports/exports (`ListBuffers`). The RM process is composed in parallel with the rest of system and synchronizes with the other processes on many actions. For example, the RM defines actions for VM creation and destruction (`INSTANTIATEVMi` (i = 1, 2, 3) and `DESTROYVM`, resp.). The RM guides also the application reconfiguration by adding and removing components from VMs (`ADDCP` and `REMOVECP`). When the RM detects a VM failure (`FAILURE`), it alerts the PS by an `ALERTPS` action.

All these actions are used for analyzing the protocol as we will see in the next subsection.

```
process MAIN [INSTANTIATEVM1:any, DESTROYVM:any,  STARTCOMPO:any,
            ADDCP:any, REMOVECP:any, ..] is
   par INSTANTIATEVM1, ..., DESTROYVM, ADDCP, REMOVECP, FAILURE,
       ALERTPS in
      (* the reconfiguration manager *)
      RM [INSTANTIATEVM1, ..., DESTROYVM, ADDCP, REMOVECP,
          FAILURE, ALERTPS] (appli)
   ||
      par AGENTtoPS1, PStoAGENT3, FAILURE, ... in
         par
            (* first virtual machine, VM1 *)
            Agent[INSTANTIATEVM1, AGENTtoPS1, PStoAGENT1,
                  DESTROYVM, STARTCOMPO, BINDCOMPO, STOPCOMPO,
                  UNBINDCOMPO, ADDCP, REMOVECP, FAILURE] (vm1)
         ||
             (* second virtual machine, VM2 *)
             Agent[...] (vm2)
          ||
              (* third virtual machine, VM3 *)
              Agent[...] (vm3)
           end par
       ||
          (* publish-subscribe messaging system *)
          PS[AGENTtoPS1, ..., PStoAGENT3, ALERTPS] (?ListBuffers)
       end par
    end par
end process
```

4.2.2 Protocol Verification

We apply the LNT specification of the protocol to a set of 600 examples (application models and reconfiguration scenarios) in order to extensively validate our protocol. When achieving these tasks, we paid much attention to cover very different examples, in particular pathological and corner cases. Hence, this large set of examples makes us very confident in the correctness of the protocol at hand. Proving the accuracy of our protocol using theorem proving techniques would be a very interesting and ambitious objective, and would complement the validation of the reconfiguration protocol using model checking.
We rely on the state-of-the-art verification tools provided by CADP to check that the protocol respects some important properties. Indeed, from the specification and an example, CADP exploration tools generate an LTS that describes all the possible executions of the protocol for each example of application model and scenario. Moreover, we specified 40 properties in MCL [102], the temporal logic used in CADP. MCL is an extension of alternation-free μ-calculus with regular expressions, data-based constructs, and fairness operators. We use model checking to verify that the properties are respected during the protocol execution. There is an exception that when a VM failure occurs, some properties are not respected (*e.g.,* a started component can be connected to a stopped one). The model checker automatically says whether these properties are not verified on the LTS. When a bug is detected by model checking tools, it is identified with a counterexample (a sequence of actions violating the property). We automated

the verification of the properties using scripts that are generated from the LNT code generator (see Fig. 4.15).

We distinguish two kinds of properties: *(i)* those verifying that the final objectives of the protocol behavior are executed (Prop. 1 below for instance) and guaranteeing that the architectural invariants for a reconfigurable application are always satisfied (Prop. 2), *(ii)* and those helping us to identify more precisely the source of error when one of the original properties was violated by verifying that the progress/ordering constraints are respected (Prop. 3, 4, 5, and 6). Following the scenarios that we want to check, these properties belong to different categories: properties dedicated to start-up scenarios (Prop. 1 and 2), destruction scenarios (Prop. 3), mixed scenarios (Prop. 4), and VM failure scenarios (Prop. 5, and 6). We present in this section some concrete properties verified on the application model showed in Figure 4.11:

1. All components are eventually started.

   (μX . (< **true** > **true and** [**not** `"STARTCOMPO !Apache1 !VM1"`] X))
   and
   ...
   and
   (μX . (< **true** > **true and** [**not** `"STARTCOMPO !MySQL !VM3"`] X))

   This property is automatically generated from the application model because it depends on the name of all VMs and components hosted on each VM.

2. A component cannot be started before the components on which it depends for mandatory imports.

   [
   true* . `"STARTCOMPO !Apache1 !VM1"` .
   true$\star$. `"STARTCOMPO !Tomcat !VM2"`
   ] false

   In the example of application showed in Figure 4.11, the Apache1 component is connected to the Tomcat component on a mandatory import.

Therefore, we must never find a sequence where Apache1 is started before Tomcat. This property is automatically generated from the application model because it depends on the component and VM names in the application model.

3. All components hosted on a VM eventually stop after that VM receives a destruction request from the DM.

   ```
   [ true* . { DESTROYVM ?vm:String } ]
       inev ( { STOPCOMPO ?cid:String !vm } )
   ```

   This property does not depend on the application. It can be verified for any application without knowing the name of VMs and components. Parameters can be related in MCL by using variables in action parameters (*e.g.*, `vm` for the virtual machine identifier and `cid` for the component identifier). This property shows the data-based features that are available in MCL.

4. There is no sequence where an import (mandatory or optional) is bound twice without an unbind in between.

   ```
   [ true* .
     "BINDCOMPO !Apache1 !Workers1" .
     ( not "UNBINDCOMPO !Apache1 !VM1" )* .
     "BINDCOMPO !Apache1 !Workers1"
   ] false
   ```

   When a component is connected to another component through an import (mandatory or optional), it cannot be bound again except if it is unbound before.

5. There is no sequence with two VM instantiations without a failure or a destroy in between.

   ```
   [ true* .
     "INSTANTIATEVM1" .
     ( not "FAILURE !VM1 or DESTROY !VM1" )* .
   ```

"INSTANTIATEVM1"

] false

When a VM is instantiated, it cannot be instantiated again except if this VM is destroyed or failed.

6. A failure action in the VM is eventually followed by an alert of this failure

   [true* . { FAILURE ?vm:String } **] inev (** { ALERTPS !vm } **)**

   This property does not depend on the application. The variable `vm` (the virtual machine identifier) used as first parameter of `FAILURE` must be the first parameter in `ALERTPS`. This property is verified for all applications and all VM names.

   This property and property 3 use the macro `inev (M)`, which indicates that a transition labeled with `M` eventually occurs. This macro is defined as follows:

   macro inev (M) = mu x .(< true > true and [not M] X)
   end macro

4.3 Experiments

We conducted our experiments on more than 600 hand-crafted examples on a Pentium 4 (2.5GHz, 8GB RAM) running Linux. Each example consists of an application model and a specific scenario (a sequence of instantiate/destroy VM operations and add/remove components to/from VMs). From this input, the CADP exploration tools generate the corresponding LTS by enumerating all the possible executions of the system. Finally, the CADP model checker is called, providing as results a set of diagnostics (true or false). The model checker returns true if a property is verified. When a property is not satisfied, it returns false as well as a counterexample. We present in this section experiments that summarize some of the numbers obtained when varying the number of VMs, the number of reconfiguration operations, and the number of failures.

We present in Figure 4.16 (left) the size of the LTS (transitions) as well as the time to execute the whole process (LTS generation and properties checking) when we modify the application model. We only instantiate VMs without destroying VMs and thus, the number of reconfiguration operations is the same as the VM number. Increasing the VMs number leads to an increase of the number of components and ports. Then, the more VMs and ports, the more parallelism in the system and, therefore, the more messages exchanged among VMs. Figure 4.16 (left) shows how the LTS size grows exponentially when we slightly increase the number of VMs in the application. The computation time scales from a few minutes for applications with 1 VM and few ports to a few hours for an application with 4 or more VMs.

Figure 4.16 (right) summarizes the results obtained for the same application used to get results showed in Figure 4.16 (left) but with a destruction and reinstantiation operations at the end of the scenario. The LTSs size and analysis time increase in a remarkable way even when just adding one destruction operation to the same application. This operation triggers a double propagation, hence more exchanged messages between the PS and the agents.

Table 4.1 shows the size of the LTS (states and transitions) before and after minimization (*wrt.* a strong bisimulation relation) as well as the time to execute the whole process in the last column of the tables (LTS generation and minimization on the one hand, and properties checking on the other). These results are obtained for the application described in Figure 4.11, with an increasing number of failures (F). The more VMs fail, the more the LTS size and analysis time increase.

Fortunately, our goal here was not to analyze huge systems (with potentially many VMs) during the verification of the protocol but to find bugs in the protocol. Indeed, most issues were found on small applications describing pathological reconfiguration cases.

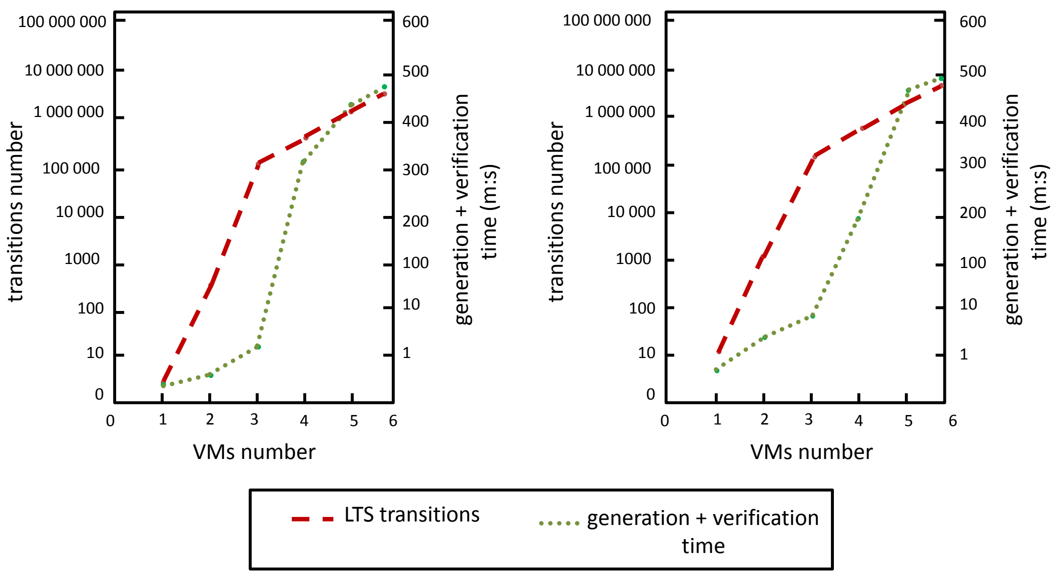


Figure 4.16 – Experimental results when increasing the number of VMs (left) start-up scenario without failure, (right) start-up and destruction scenario without failure

4.4 Problems Found and Corrected

We have presented in Section 4.2 the specification and verification of our protocol. Model checking tools permitted us to find bugs that were identified by counterexample analysis. This allowed us to revise several parts and correct specific issues (*e.g.*, adding some acknowledgement messages after binding/unbinding ports, starting/stopping components, etc) in both the specification and implementation that were written at the same time. During the verification steps, we detected two important issues that we corrected in the latest version of the protocol. The implementation was systematically corrected. In the remainder of this section, we focus on these two problems.

F	LTS (states/transitions)		Time (m:s)
	raw	minimized	LTS gen. / Verif.
0	452,378/983,963	8,531/29,394	9:21 / 1:84
1	612,293/1,262,732	9,568/31,472	10:37 / 1:79
2	682,459/1,420,543	12,390/38,971	16:59 / 2:83
3	793,813/1,584,238	16,673/48,562	24:73 / 3:69
4	993,527/1,763,227	19,586/63,254	31:91 / 4:56

Table 4.1 – Experimental results (with failures)

In the first version of the protocol, the RM was centralized and in charge of creating VMs, starting and stopping components. Therefore, it kept track of the states of components for each VM. To do this, the RM was informed every time there was a change in the application. It communicated with all the agents by exchanging messages in order to update the application after each change in the component architecture (*e.g.*, a component binds to another component, a component changes its state). There was an overhead of messages transmitted to and from the RM. We noticed during our experiments that even with simple applications, CADP generated huge LTSs. We solved this drawback by proposing a decentralized version of the protocol. The new version of the protocol consists of an RM that is not in charge of starting or stopping components. The RM guides only the application reconfiguration by instantiating, destroying, and repairing VMs, or by adding and removing components from an existing VM. It is part of the agent behavior to drive the component start-up/shutdown. The decentralized version of the protocol avoids additional, unnecessary messages exchanged between agents and the RM. This version guarantees more parallelism and better performance in the corresponding implementation of the protocol.

The second issue that we detected during the verification of the protocol is in the way VMs are destroyed. Originally, when a component required to stop, it was stopped and then all components bound to it were stopped. Stopping components in this order induced started components connected to stopped components. This violated the consistency of the component composition and wellformedness architectural invariants. This bug was detected thanks to a property stating that *" a component cannot be started and connected through an import (mandatory or optional) to a stopped component "*. Thus, we corrected this bug by proposing another way to stop components. When a component needs to stop, it requests to all components bound to it to unbind, and once it is done, it can immediately stop. This implies first a backward propagation along components bound on mandatory imports. Once this first propagation stops (a component does not provide service or is connected only through optional imports), we start a forward propagation during which components are actually stopped and indicate to their partners that they have just stopped and unbound. This double propagation, as presented in Section 4.1.4.2, is necessary for preserving the component architecture consistency and for avoiding that started components can keep on using stopped components.

4.5 Protocol Implementation

The protocol was implemented by another partner of the OpenCloudware project in RoboConf [114]. Roboconf is a system for configuring, installing, and managing complex legacy application stacks deployed on the cloud which dynamically evolve in time[3]. We present in this section some implementation details concerning the protocol. The core of the system of our implementation is roughly 6K lines of Java code. It is based on an IaaS abstraction that allows to instantiate VMs on different IaaS layers (such as Amazon EC2 and Microsoft Azure) and on RabbitMQ[4], which is an implementation of the AMQP standard[5]. It is a message server, implementing the publish-subscribe messaging system. It is mainly used for the communication between VMs and RM. An AMQP communication system was chosen in order to have a reliable and asynchronous communication system.

When the user requests the deployment of a new software instance, the RM first contacts the IaaS API to check about the VM instance defined in the soft-

3. http://roboconf.net/fr/index.html
4. http://www.rabbitmq.com/
5. http://www.amqp.org

ware instance model. If the user asks to deploy the software application on an already existing VM instance, the RM immediately sends the model to the VM. Otherwise, the RM checks if the software type of the software instance has a VM type defined. If so, the RM asks the IaaS to instantiate one VM of this kind. Otherwise, the RM asks the IaaS to instantiate a VM from the default VM template. When the RM asks the IaaS to instantiate a VM, it creates a message buffer on the message server (implementing the publish-subscribe messaging server) for that VM.

When the RM sends the software instance model to the VM instance, it serializes the software instance object and sends it to the message buffer of the VM along with the configuration files (Puppet recipes for example). This part can be done even when the VM is not running thanks to the messages stored in its message buffer. When the VM boots, the agent starts, connects to the message buffer and gets the message.

The agent on the VM gets the software instance model and the configuration files associated with it. It checks what "connector" it has to use to perform the operations for the software instance. A connector is a Java class that implements the four operations: setup, update, start, and stop. A connector is independent from any software it will install, it only does basic operations. The operations related to the software are located in the configuration files. Each connector, when calling a basic operation, transmits the configuration of the model (such as variables and imported variables) to the packaging and configuration system. This mechanism enables the user to separate actual operations on VM from binding details. Multiple connectors enable users to keep the original way of installing and configuring the software. In addition, it also enables to cover almost any kind of software: from software available in Linux repositories to legacy software. A Puppet[6] connector was implemented in order to install and update software on VMs. Puppet is one of the most known configuration systems. It provides a language and a program. The language enables to describe the state of a system by describing its packages, files, and services.

When the installation is completed (software instances are installed on VMs), each agent publishes its configuration (according to the exports) and sub-

6. http://docs.puppetlabs.com/

scribes to other remote configurations (according to the imports) by using the RabbitMQ API.

4.6 Conclusion

We have successively presented in this chapter the design, specification, and verification of a novel reconfiguration protocol involving components distributed over several VMs. This protocol enables one to instantiate new VMs/destroy existing VMs and add a new component/remove an existent one. Upon reception of one of these reconfiguration operations, each VM agent connects/disconnects and starts/stops components in a particular order in order to preserve the application consistency (*i.e.,* architectural invariants), which is quite complicated due to the high parallelism degree of this protocol. The protocol also supports the VMs failures. It detects failures and repairs them by creating new instances of the failed VMs. We specified and verified our protocol using the LNT specification language and the CADP toolbox, which is considered as very convenient for modeling and analyzing such protocols. We used model checking techniques to verify 40 properties of interest on a large number of application models and reconfiguration scenarios. The experience was successful because we detected several issues that were corrected in the corresponding Java implementation, which is established by another partner of the OpenCloudware project. In particular, we deeply revised the part of the protocol dedicated to the VM destruction and component shutdown.

5
Asynchronous Coordination of Stateful Autonomic Managers in the Cloud

" The value of achievement lies in the achieving. "

Albert Einstein

Contents

THis thesis proposes new techniques that aim at automatically monitoring and updating complex distributed applications. Managing these applications is a challenging problem because the manual administration is no longer realistic for complex systems. Thus, autonomic computing is a promising solution for automating the administration functions, which focus particularly on replicating virtual machines, destroying or adding them, and handling VM failures in the cloud. These operations are executed by different autonomic managers considered as control loops. Each manager observes the application execution, ensures a continuous monitoring, and immediately reacts to changes by automatically executing reconfiguration tasks. Several managers can be deployed to manage the same application and must make consistent decisions. Nonetheless, using them without coordination may lead the system into inconsistencies and error-prone situations (*e.g.,* removing a server that will be needed). As a consequence, the use of multiple managers (*e.g.,* self-repair and self-sizing managers) implemented in the same system requires taking globally consistent decisions. Hence, a manager should be aware of decisions of all managers before reacting. We present in this chapter our synthesis techniques for generating a controller, which aims at coordinating several managers. The generated controller prevents every manager from violating global objectives of all the managers. Our controller synthesis techniques ensure an asynchronous interaction of all the managers with the generated controller, meaning that all the messages transmitted from/to the managers (controller, resp.) are stored/-consumed into/from FIFO buffers.

More precisely, an autonomic manager is described using a formal model, namely a Labelled Transition System (LTS). We defined a set of reaction rules and regular expressions to specify the coordination requirements and interaction constraints. As a consequence, each manager is not only able to manage its internal behaviour but also its relationship with other autonomic managers, which is achieved by the specification of the coordination requirements. Looking at Figure 5.1, one can observe that our controller synthesis techniques rely on an encoding of our inputs (LTS models and coordination requirements) into the LNT process algebra. We particularly used the CADP compilers and minimization tools (*e.g., strong equivalence*) to generate a reduced LTS from the LNT specification. The generated LTS corresponds to all possible executions of the controller. It is worth noting that since we rely on formal techniques and tools, all the verification techniques available in the CADP toolbox can be used for validating the generated controller.

Once we have synthesized the controller LTS and we are satisfied by the coordination requirements, Java code is generated using a code generator we developed. This Java code is finally deployed and used for coordinating real applications. In this chapter, we use a typical example of a N-tier Web application as running example. We validated our approach on several variants of this distributed application involving several instances of autonomic managers, such as self-sizing or self-repair managers.

We emphasize that our approach covers the whole development process from the expression of the requirements to the final implementation and deployment of our solution.

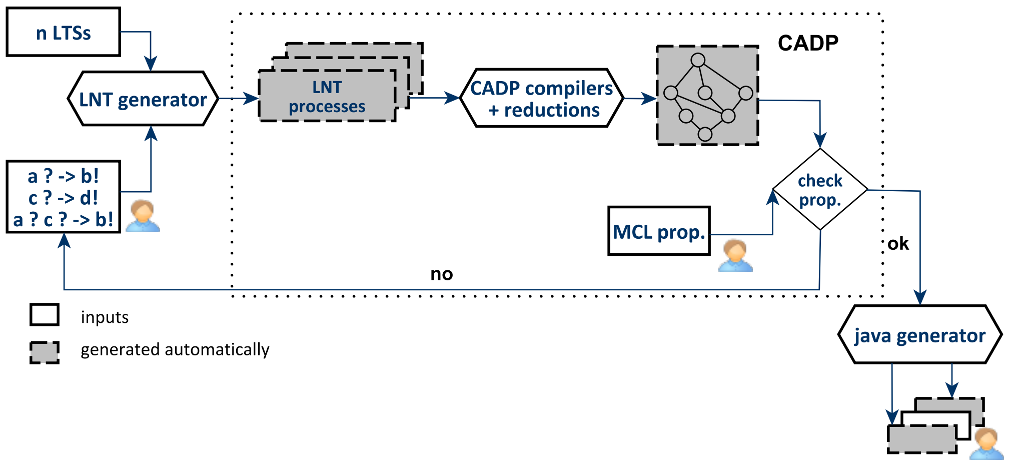


Figure 5.1 – Overview of our approach

5.1 Formal Models

In this part, we first introduce the abstract model that we used to represent an autonomic manager. Second, we propose reaction rules and regular expressions for specifying how the involved managers are supposed to interact together through the controller that we want to generate. The managers models and the coordination expressions are used as inputs to our synthesis technique (see Section 5.2).

5.1.1 Autonomic Manager Model

Each autonomic manager is modelled as a Labelled Transition System (LTS), which consists of a set of states and transitions that occur between those states. Formally, an LTS is defined as follows:

Definition 4. *A Labelled Transition System is a tuple defined as $LTS = (Q, A, T, q^0)$ where Q is a finite set of states, $A = A^! \cup A^?$ is an alphabet partitioned into a set of send and received messages, $T \subseteq Q \times A \times Q$ is a transition relation, and $q^0 \in Q$ is the initial state.*

We refer to a send message $m \in A^!$, which must be required and consumed by another manager or by the controller, as $m!$. We refer to a received message $m \in A^?$, which is consumed by the actual manager, as $m?$. A transition that occurs between two states is represented as $q \xrightarrow{l} q' \in T$ where $q, q' \in Q$ and $l \in A$. We assume that managers are deterministic, which can be easily obtained using standard determinization algorithms [79].
Given a set of manager LTSs, where every manager is defined as (Q_i, A_i, T_i, q_i^0), we assume that each message should have a unique sender and a unique receiver: $\forall i, j \in 1..n$, $i \neq j$, $A_i^! \cap A_j^! = \emptyset$ and $A_i^? \cap A_j^? = \emptyset$. Furthermore, each message is exchanged between two different managers: $A_i^! \cap A_i^? = \emptyset$ for all i. The uniqueness of messages can be achieved via renaming.

5.1.2 Coordination Requirements Specification

The backbone of our approach is the coordination of multiple managers, handling the same system, using a generated controller. To do this, we use reaction rules and regular expressions with their basic operators, such as sequence, choice, and iteration, to describe the behaviour one expects from the controller.

The generated controller aims at orchestrating the execution of the managers. We define a reaction rule as a set of receptions followed by a set of emissions. Basically, it expresses that when the controller receives a set of messages from managers within a certain period of time (left hand part), it must send all the messages specified in the second set (right hand part) once the period is expired. We note that the real period will be chosen during the deployment phase, and both sets of messages can be received and emitted in any order.

Definition 5. *Given a set of managers $\{M_1, \ldots, M_n\}$ with $M_i = (Q_i, A_i, T_i, q_i^0)$, a reaction rule R is defined as $a_1, ..., a_m \rightarrow b_1, ..., b_p$ where a_j presents a message received from a manager M_i ($a_j \in A_i^?$) and b_k presents a message emitted to a manager M_l ($b_k \in A_l^!$) for $1 \leqslant j \leqslant m$ and $1 \leqslant k \leqslant p$.*

The specification of the behaviour one expects from the controller is expressed using a coordination expression C.

Definition 6. *A coordination expression C is a regular expression over reaction rules R:*

$$C ::= R \mid C_1.C_2 \mid C_1 + C_2 \mid C*$$

where $C_1.C_2$ represents a coordination expression C_1 followed by C_2, $C_1 + C_2$ represents the choice between C_1 and C_2, and $C*$ presents a repetition of C zero or several times.

It is worth noting that all participants, namely the autonomic managers and the controller to be generated, communicate asynchronously using message passing via FIFO buffers. Each participant is equipped with one input buffer. Therefore, it consumes messages from its buffer and sends messages that are putted at the input buffer of the message's recipients. Once generated and added to the system, all managers communicate through the controller, which means that the controller acts as a centralized orchestrator for the whole system.

5.2 Asynchronous Synthesis Techniques

In this section, we present new asynchronous controller synthesis techniques, which rely on an encoding of our models (aut format[1]) and coordination requirements into the LNT specification language. From this LNT specification, we can generate the corresponding LTS model of the controller using CADP compilers, hiding, and reduction techniques. We also use CADP verification tools to validate the generated controller. All the steps presented in this section are fully automated using a tool that we developed in Python[2]. This tool generates the LNT code as well as SVL scripts that are used for invoking CADP exploration and reduction tools, which finally results in the generation of the controller LTS.

5.2.1 Process Algebra Encoding

We present successively in this section the encoding into LNT of the different parts of our system, *i.e.*, autonomic managers, coordination requirements, and architecture.

5.2.1.1 Encoding of An Autonomic Manager

An LNT process is generated for each state in the manager LTS. Each process is named using the state identifier of the corresponding state. The alphabet of the process consists of the set of send and receive messages appearing on the LTS transitions. The behavior of the process encodes all the transitions of the LTS going out from the corresponding state and eventually calls the processes encoding the target states of those transitions. Actually, if there is no such transition, the body of the process is the `null` statement. If there is a single transition, the body of the process corresponds to the message labelling this transition, followed by a call to the process encoding the target state of the transition. Finally, if there is more than one transition, we use the `select` operator. Let us assume that two transitions $\mathtt{q}\xrightarrow{l}\mathtt{q'}$, $\mathtt{q}\xrightarrow{l'}\mathtt{q''} \in T$ have the same source state `q`. The behavior of the process encoding `q` in LNT is:

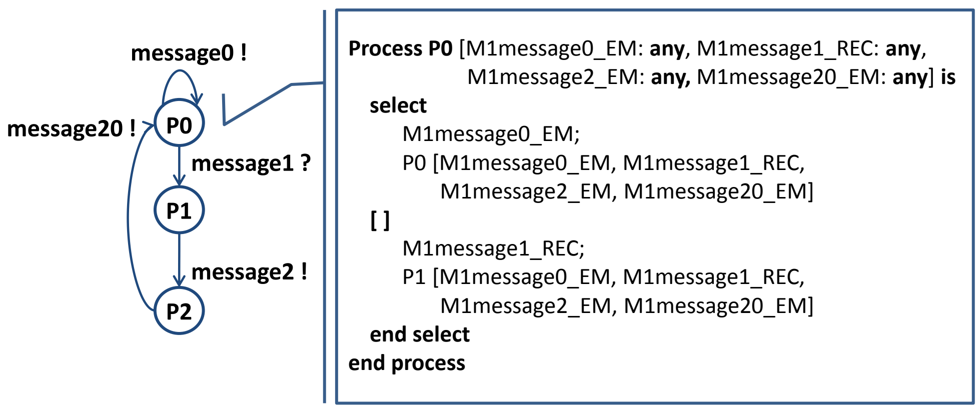
1. The aut format is the textual format used to represent automata in the CADP toolbox, see http://cadp.inria.fr/man/aut.html

2. We developed an LNT code generator in Python for automating this task.

select `l; q' [...]` **[]** `l'; q"[...]` **end select**

where the LNT operator `select` encodes a non-deterministic choice between `l` and `l'`.

Since a message name can be used in different autonomic manager LTSs, each message is prefixed with the manager name to avoid further name clashes. We encode emitted messages (received messages, resp.) with a `_EM` (`_REC`, resp.) suffix. These suffixes are necessary because LNT symbols ! and ? are used for the data transfer only. As an example, $m1 \in A^!$ is encoded as $m1\_EM$, and $m2 \in A^?$ is encoded as $m2\_REC$.



```
Process P0 [M1message0_EM: any, M1message1_REC: any,
            M1message2_EM: any, M1message20_EM: any] is
  select
     M1message0_EM;
     P0 [M1message0_EM, M1message1_REC,
        M1message2_EM, M1message20_EM]
  [ ]
     M1message1_REC;
     P1 [M1message0_EM, M1message1_REC,
        M1message2_EM, M1message20_EM]
  end select
end process
```

Figure 5.2 – Example of a manager modelled as an LTS (left), and the LNT process encoding the initial state (right)

Figure 5.2 exhibits the LTS model of a manager example, as well as the encoding of its initial state `P0` into the LNT process algebra. We observe that the process name is the same as the identifier of that state, whereas the process alphabet is composed of all the messages appearing in the LTS, prefixed with the name of the manager (`M1`) and suffixed by $\_EM$ in the case of an emission and by $\_REC$ in the case of a reception.

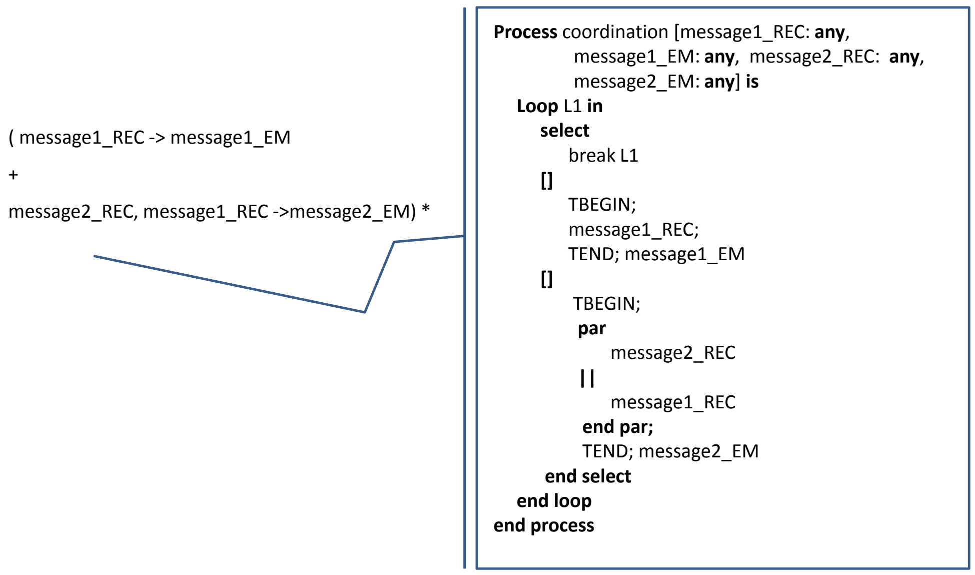
5.2.1.2 Encoding of the Coordination Requirements

The coordination requirements specified using reaction rules and regular expressions describe an abstract version of the controller to be generated. These requirements are encoded into an LNT process called *coordination*. The process alphabet is composed of all received and emitted messages appearing in the reaction rules. The body of this process encodes the regular expression of reaction rules.
Each reaction rule is translated to LNT separating both sides of the rule (*i.e.*, left hand part and right part) using the sequential composition construct (`;`). In order to make explicit in the controller LTS the logical interval of time that will be chosen in the implementation step and during which the controller receives messages, the left hand part of the reaction rule starts with an action `TBEGIN` and ends with an action `TEND`. This part is translated using the `par` operator without synchronization (pure interleaving) since all messages can be received in any order (see Fig. 5.3). After the execution of the `TEND` action, the right hand part of the reaction rule is translated using the sequential composition. We will explain in the Section 5.2.2 why this part is not translated using the parallel composition without composition, as is the case for the left hand part.
As far as the regular expression is concerned, a sequence (.) of rules is encoded using the sequential composition (`;`), a choice (+) between several rules is translated using the **select** construct, and an iteration ($*$) is encoded using the **loop** operator as follow:

```
loop L1 in
    select break L1 [] ... end select
end loop
```

5.2.1.3 Encoding of the Architecture

In this section, we present how all participants, *i.e.*, managers and coordination expression, are composed altogether. The communication between them is achieved asynchronously. The coordination expression represents an abstract description of the future controller that we aim to generate, and all messages



```
( message1_REC -> message1_EM
+
message2_REC, message1_REC ->message2_EM) *
```

```
Process coordination [message1_REC: any,
          message1_EM: any,  message2_REC:  any,
          message2_EM: any] is
   Loop L1 in
      select
         break L1
      []
         TBEGIN;
         message1_REC;
         TEND; message1_EM
      []
          TBEGIN;
          par
              message2_REC
          ||
              message1_REC
          end par;
          TEND; message2_EM
       end select
   end loop
end process
```

Figure 5.3 – Example of coordination requirements encoded into an LNT process

must go through this controller, which acts as a centralized orchestrator. Each participant is equipped with an input FIFO buffer. When a participant wants to read a message, it consumes the oldest message putted in its buffer. When it sends a message to another participant, the message is stored in the input buffer of that participant. LNT functions are used to describe basic operations on these buffers (*e.g.*, adding and retrieving messages). We present below, an example of function that removes a message from a FIFO buffer (*i.e.*, from the beginning).

```
function remove_MSG (q: TBUFFER): TBUFFER is
  case q in
     var hd: TMessage, tl: TBUFFER in
          nil         -> return nil
        | cons(hd,tl) -> return tl
  end case
end function
```

The function takes as input a buffer (`q`), which type is (`TBuffer`) and composed of a list of messages (`TMessage`). If the buffer is empty, nothing happens. If it is not empty, the first message is removed.

It is worth noting that our synthesis techniques allow one to choose buffer bounds. One can either decide to fix an arbitrary bound for buffers or to use unbounded buffers. In the first case, the only constraint is that the same buffer bound should be used when deploying the controller. Otherwise if at some point it must be changed (*e.g.,* after a modification in memory requirements), unexpected behaviours and erroneous situations may occur. In the second case (unbounded buffers), the risk is to attempt to generate a controller whose corresponding state space is infinite [27]. As an intermediate solution, one can use the recent results presented in [19] for identifying whether the interactions between managers with unbounded buffers can be mimicked with bounded buffers. If this is the case, the lower bound returned by these techniques should be used as the minimum buffer bound for both our synthesis techniques and the deployment of the application.

A buffer in LNT is first encoded using an LNT list and classic operations on it. Then, for the behavioral part, a buffer is encoded using a process with a buffer data type as a parameter. This process can receive messages from the other participants, and synchronizes with its own participant when that one wants to read a message. We generate a process encoding each couple $(participant, buffer)$ that corresponds to a parallel composition (`par`) of the participant with its buffer. The synchronization set contains messages consumed by the participant from its buffer.

Finally, the whole system (`main` process in LNT, see below) consists of the parallel composition of the couple $(coordination, buffer)$ and the couples $(manager_i, buffer_i)$ generated for all the managers. It is worth noting that since autonomic managers communicate via the controller, they evolve independently from one another and are therefore composed using the `par` operator without synchronizations. In contrast, the couple $(coordination, buffer)$ must synchronize with all the other couples on all emissions from the managers/to the buffers, and this is made explicit in the corresponding synchronization set of this parallel composition.

```
process main [message1:any, ..., messagen:any] is
   par messagep, ..., messagek in
     couple_buffer_coordination [...]
```

```
      ||
       par
         couple_buffer_manager$_1$ [...]
       || . . . ||
         couple_buffer_manager$_n$ [...]
      end par
    end par
end process
```

5.2.2 Compilation and Verification

5.2.2.1 Generation of the Controler LTS

Once all the inputs (models and coordination requirements) are encoded into LNT, we can use compilers available in the CADP tools to obtain the LTS corresponding to all behaviors of the LNT specification. In order to keep only the behavior corresponding to the most permissive controller [94, 137], we need to hide message exchanges corresponding to consumptions of the managers from their buffers and emissions from managers to the coordination expression buffer. All these messages are replaced by internal actions τ. Thus, we use the reduction techniques available in CADP for getting rid of internal actions, removing duplicated paths, and determinizing as much as possible the final LTS. In this context, the reduction can be ensured on-the-fly modulo: (1) strong equivalence for replacing duplicate transitions by a single transition and (2) weak trace equivalence for determinizing the generated LTS. Finally, we preserve only local emissions/receptions from the coordination expression point of view (messages shown in the dashed grey rectangle in Fig. 5.4). Transitions figuring in the final LTS are labelled with the messages corresponding to the process alphabet of the couple $(coordination, buffer)$.

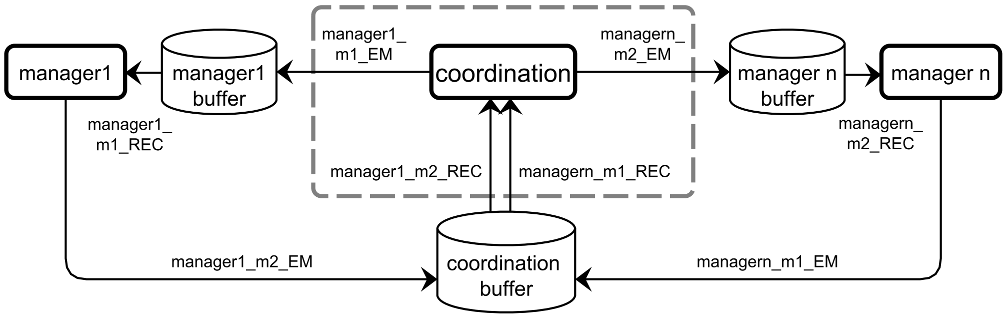


Figure 5.4 – Exchange of messages between the coordination expression and the managers

5.2.2.2 Verification of the Generated Controller

Last but not least, let us stress that, since the writing of the coordination expression is achieved manually by a designer, this step of our approach may lead to an error-prone expression. Indeed, the designer can provide a wrong specification of the coordination requirements, which does not take into consideration the global objectives of all the managers. However, we can take advantage of the encoding into LNT to check either the controller LTS alone (and thus the coordination expression) or the LTS corresponding to the final system (*i.e.*, composition of all participants). To do so, one can use the CADP model checker (Evaluator), which takes as input an LTS model and temporal properties specified in MCL [102]. We distinguish two types of properties:

1. properties which depend on the application (*e.g.*, the controller must eventually transmit a specific message to a certain manager);
2. properties which do not depend on the application (*e.g.*, checking the absence of deadlock). One of these properties was specified at the deployment step and allowed us to avoid the nondeterministic choice between different behaviors which can be executed by the controller once the period is expired. This property allows us to correct the coordination requirements, more precisely, the right hand part of the reaction rules. It verifies that the `TEND` action should never be followed by a choice between a set of possible executions, meaning that the left hand part of every reaction rule must be followed by only one behavior.

5.3 Java Code Generation and Deployment

We present in this section our techniques to automatically generate the Java code, which corresponds to the controller LTS obtained during the compilation phase (see Section 5.2.2). This Java code allows to deploy controllers in the context of real-world applications.

5.3.1 Java Code Generation Techniques

Our Java code generation techniques are based on the use of object-oriented programming. It take as input the controller LTS synthesized beforehand and automatically generate all Java classes, methods, and types necessary for deploying it. The controller LTS is encoded as an instance of a Java class `LTS`. This class relies on two classes, namely a class `State` and a class `Transition`, which represents the transitions between the states. The `LTS` class also defines an attribute `cstate` representing the current active state in the controller model. This variable is initialized with the LTS initial state. Some Java code is necessary to interface the controller with the running application. We particularly define a method called `react` that takes as input a list of messages received within a period of time and applies successive moves according to the received messages, the current state of the controller, and the behavior of the generated controller. This method computes the messages that the controller has to send as a reaction to these received messages and updates the current state of the controller.

5.3.2 Deployment

Our generated Java code can be deployed and applied on concrete applications using the event-based programming paradigm. The period of time described using special actions `TBEGIN` and `TEND` in the controller LTS has to be instantiated with a real value. This period is computed using sampling techniques and implemented using the `sleep` method in Java. The choice of this period cannot be realized during the synthesis phase and is achieved just before deployment.

The main behaviour of the controller (`run` method) consists of an infinite reactive loop, which successively receives events from the application and computes reactions (messages to be sent by the controller). The result produced by each call of the `run` method (*i.e.*, controller reactions) must also be interpreted and executed. Therefore, all the messages returned by that method are encoded as events too. A part of the Java program is dedicated to handling the events raised by the application by converting them into the input format of the `react` method, and conversely handling the output of the `react` method by translating it into a list of events executed by the system. Each event contains the corresponding message and additional information, for instance, a failure event also has as parameter the impacted server and further information (identifier, port, etc.). Therefore, all the messages received/emitted by the generated controller are connected to the corresponding events.

5.4 A Multi-tier Application Supervised by Autonomic Managers

We introduce in this section an example, which consists of a JEE multi-tier application supervised by instances of two sort of autonomic managers, namely a self-repair and a self-sizing manager. We also present an example of coordination requirements describing how to orchestrate several instances of these autonomic managers.

5.4.1 Multi-Tier Application

Our example is a JEE multi-tier application (Fig. 5.5) composed of an Apache Web server, a set of replicated Tomcat servers, a MySQL proxy server, and a set of replicated MySQL databases. The Apache server receives incoming requests and distributes them to the replicated Tomcat servers. The Tomcat servers access the database through the MySQL proxy server that distributes the SQL queries to a tier of replicated MySQL databases fairly.

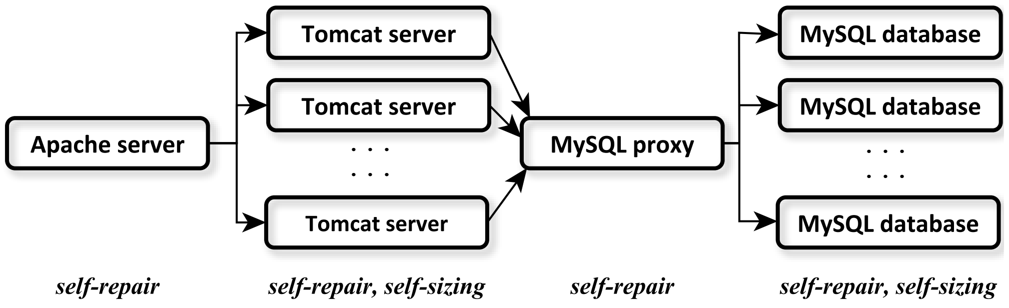


Figure 5.5 – A multi-tier application

5.4.2 The Managers Role

The dynamic sizing plays an important role in the energy consumption of the replicated hosted on this type of application. Varying the number of requests makes difficult the estimation of the number of duplicated servers when starting the application. Therefore, adjusting dynamically the degree of replication allows to allocate only the number of needed servers according to the number of requests. Moreover, once a server fails, the failure must be immediately detected and repaired in order to preserve the application consistency. To optimize the energy consumption of the replicated servers and preserve the application consistency, we used managers such as self-sizing and self-repair for the management of the dynamic sizing and the servers repair of multi-tier applications. Figure 5.6 displays how the architecture of each one of these managers is based on the `MAPE-K` (Monitor Analyse Plan Execute - Knowledge) reference model (see Section 2.4). For instance, the self-sizing manager continuously observes a load of replicated servers through the Monitor function. It computes the average of the load and detects the presence of an overload or an under-load through the Analyze function. Once an overload (under-load, resp.) is detected, the manager makes a decision about the addition or removal of a server through the Plan function. This decision is executed by the execute function.

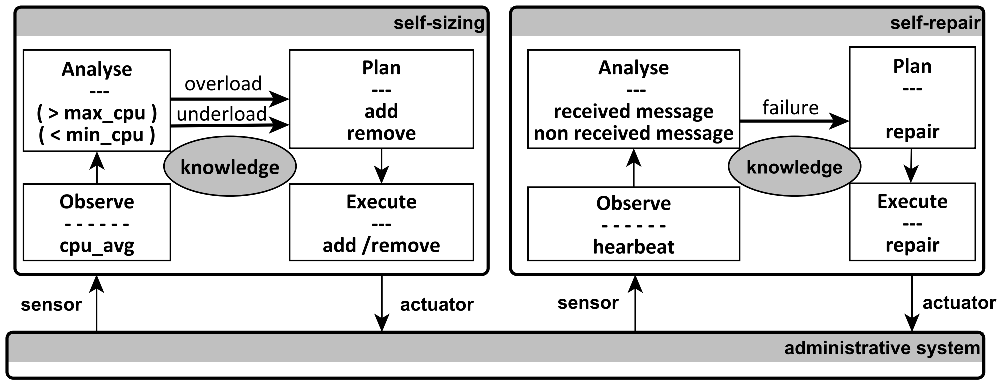


Figure 5.6 – (left) The architecture of the self-sizing manager, (right) the architecture of the self-repair manager

5.4.3 The Managers Models

We describe the managers behaviors using several LTS models. First, we model the behavior of the monitor, analyse, and execute functions of the managers by what we call the application manager (Fig. 5.7, right), which sends messages when a change occurs in the system and receives messages indicating actual administrative changes to perform on the application. As for the plan functions, we use two models called self-sizing and self-repair managers, resp. More precisely, the generated controller has the role of coordinating the messages transmitted from the analyze function to the plan function and from the plan function to the execute function.

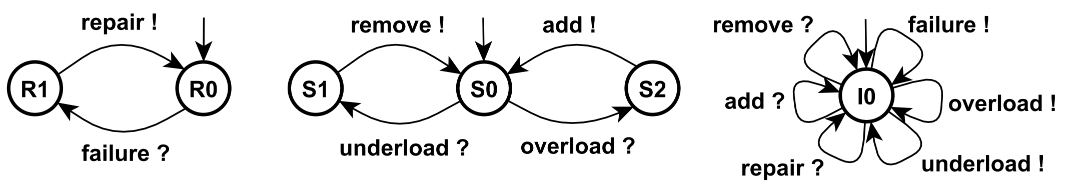


Figure 5.7 – (left) Self-repair manager LTS, (middle) Self-sizing manager LTS, (right) Application manager LTS

5.4.3.1 The Self-Sizing Manager

The self-sizing is in charge of adapting the number of replicated servers dynamically by sending the message `add!` (`remove!`, resp.) to the system when detecting an overload (underload, resp.). The overload (underload, resp.) is detected when the average of the load exceeds (is under, resp.) a maximum (minimum, resp.) threshold (Fig. 5.7, middle). We associate one instance of the self-sizing manager to the Tomcat servers and another instance to the MySQL databases. Therefore, we consider, here, that we have two managers having two different models. Both models have the same states and transitions as the LTS shown in Figure 5.7 (middle), but suffixed by `_tc` (`_mq`, resp.) when the instance is associated with the Tomcat servers (MySQL databases, resp.).

5.4.3.2 The Self-Repair Manager

The self-repair manager asks the system to repair a failure by creating a new instance of the failed server (Fig. 5.7, left). We have four instances of the self-repair manager, one per tier. Therefore, we consider, here, that we have four managers. All the models have the same states and transitions as the LTS shown in Figure 5.7 (left), but suffixed by `_ap` (`_tc`, `_pq`, `_mq`, resp.) when the instance is associated to the Apache server (Tomcat servers, MySQL proxy server, MySQL databases, resp.).

5.4.4 Coordinated Problems

The absence of coordination between these managers may lead the whole system to some undesired situations such as adding two new servers whereas one was enough or removing a server that is needed, as result of a server failure. We distinct two types of problems: those occurring in the same tier (*i.e.,* the replicated servers) and those occurring in different tiers.

5.4.4.1 Coordinated Problems in the Same Tier

A failure of a server in a set of replicated servers triggers an overload of the remaining servers. Actually, when the self-repair manager receives a message indicating the detection of this failure, it sends a message to the application manager requesting it to repair the failure by creating a new replica. Before completion of this repair phase, the other replicated servers receive more requests than before the failure, which causes an overload. Upon reception of this overload detection from the application manager, the self-sizing manager sends a message back asking the addition of another server. In this scenario, as a result of a server failure, two new servers are added to the application whereas one was enough.

5.4.4.2 Coordinated Problems in Different Tiers

A failure of a server, which is hosted on a first tier and connected to other servers hosted on another tier, triggers an under-load in the second tier. Actually, when the self-repair manager receives a message indicating the detection of this failure, it sends a message to the application manager requesting it to repair the failure by creating a new replica. Before completion of this reparation, the servers hosted on the second tier receive fewer requests than before the failure, which causes an under-load. Upon reception of this under-load detection from the application manager, the self-sizing manager sends a message back calling for the removal of a server. Then, once the failed server is repaired, the servers hosted on the second tier receive more requests than before the server reparation, which causes an overload and therefore the addition of another server by the self-sizing manager. Therefore two unnecessary operations (*i.e.,* the removal and the addition operations) are executed as a result of a server failure.

5.4.5 Managers Coordination

We present below an excerpt of the requirements for the controller we want to generate for our example. These rules ensure that all managers globally satisfy the coordination objectives. Each line presents the actions that can be received by the controller in a period *T* (left parts of reactions rules). At the end of each period, if the received messages match the left part of one fireable rule, it reacts by emitting the messages appearing in the right part of that rule. All messages are prefixed by the manager name (`app` stands for the application manager) and suffixed by the name of the tier to which is associated the manager.

```
 ( app_failure_ap?    -> repair_failure_ap!                     (❶)
 + app_failure_tc?    -> repair_failure_tc!                     (❷)
 + app_overload_tc?   -> sizing_overload_tc!                    (❸)
 + app_underload_tc?   -> sizing_underload_tc!                  (❹)
 + app_failure_mysql? -> repair_failure_mysql!                  (❺)
 + app_failure_px?    -> repair_failure_px!                     (❻)
 + app_failure_ap?, app_underload_tc? -> repair_failure_ap! (❼)
 + app_failure_tc?, app_overload_tc?  -> repair_failure_tc! (❽)
 + ...  ) *
```

We distinguish two kinds of rules:

1. those where a unique message appears in the left part of the reaction rule (see, *e.g.,* ❶, ❷). In that case, the corresponding controller immediately transfers that message to the manager;
2. those encoding the coordination we want to impose on managers, *e.g.,* rule ❽ permits to generate a controller that can avoid to add two Tomcat servers by forwarding only one of the two received messages on a same period of time.

Last, since there is no specific order between all these rules, we use a simple regular expression where all rules can be fired at any time (combination of + and * operators).

5.4.6 Encoding into LNT

Let us first show some excerpts of LNT obtained when calling our LNT code generator in this example. We first show the LNT processes encoding an instance of the repair manager, which handles the tier of the Tomcat servers. The first process has the same name as the initial state identifier

`R0_tc`. The process alphabet is the set of labels used in this manager LTS (`repair_failure_tc_REC` and `repair_repairing_tc_EM`). Each message is prefixed with the manager name. The body of the first process, for instance, receives a failure message (`repair_Failure_tc_REC`) and calls the process encoding the target state, that is `R1_tc`.

```
process R0_tc [repair_failure_tc_REC: any,
               repair_repairing_tc_EM: any] is
     repair_failure_tc_REC;
     R1_tc [repair_failure_tc_REC, repair_repairing_tc_EM]
end process

process R1_tc [repair_failure_tc_REC: any,
               repair_repairing_tc_EM: any] is
      repair_repairing_tc_EM;
      R0_tc [repair_failure_tc_REC, repair_repairing_tc_EM]
end process
```

We show now an example of process encoding a couple $(manager, buffer)$, particularly the couple corresponding to the repair manager handling the tier of the Tomcat servers. This manager synchronizes with its buffer on the `repair_failure_tc_REC` message, which is emitted by the buffer and received by the manager. Note that the buffer process (`buffer_repair_tc`) is equipped with a parameter corresponding to the buffer data type, that is the structure where messages are stored, initialized to nil.

```
process couple_buffer_repair_tc [repair_failure_tc_REC: any,
      repair_repairing_tc_EM: any, repair_failure_tc_EM: any] is
  par repair_failure_tc_REC is
    R0_tc [repair_failure_tc_REC, repair_repairing_tc_EM]
  ||
    buffer_repair_tc [repair_failure_tc_EM, ...] (nil)
  end par
end process
```

After generating all the couples (autonomic manager/buffer, application manager/buffer, and controller/ buffer), the main process is encoded using parallel compositions. The managers do not interact directly together. Therefore their couples are translated using the `par` construct without synchronizations (pure interleaving). All the managers communicate together through the controller. This interaction is expressed using another parallel composition where the synchronization set makes explicit all emissions sent by the managers to the controller buffer, or sent by the controller to the managers buffers.

```
process main [repair_failure_ap_REC:any, ...] is
   par sys_failure_ap_EM, sys_failure_tc_EM,  ... in
     (* couple coordination/buffer  *)
        couple_buffer_coordinaiton [...]
     ||
     par
       (* couple application manager/buffer *)
        couple_buffer_AM [...]
     ||
     (* couple repair manager/buffer for the Apache server*)
        couple_buffer_repair_ap [...]
     ||
     (* couple repair manager/buffer for the Tomcat servers*)
        couple_buffer_repair_tc [...]
     ||
     (* couple repair manager/buffer for the MySQL proxy server*)
        couple_buffer_repair_px [...]
     ||
     (* couple repair manager/buffer for the MySQl databases*)
        couple_buffer_repair_mq [...]
     ||
     (* couple sizing manager/buffer for the Tomcat server *)
        couple_buffer_sizing_tc [...]
     ||
     (* couple sizing manager/buffer for the MySQL databases *)
        couple_buffer_sizing_mq [...]
    end par
   end par
end process
```

5.4.7 Compilation and Verification

From the encoded LNT specification obtained when calling the LNT code generator, we use CADP compilers to generate the LTS describing the whole system for our running example (consisting of 194,026,753 states and 743,878,684 transitions). Then, we use hiding and minimization techniques to generate the LTS of the controller (consisting of 28,992,305 states and 46,761,782 transitions). An excerpt of the controller LTS, which focuses on the failure and overload detection of a Tomcat server in the same period of time, is shown in Figure 5.8. We recall that we use specific labels (namely `TBEGIN` and `TEND`) for characterizing the messages received during a same period of time. This LTS shows that when the controller receives a failure and an overload message (of a Tomcat server in this example) during a same period, it forwards only the failure message and drops the overload message. In contrast, when the controller receives these two messages in two different periods, it forwards them to the repair and sizing manager, resp.

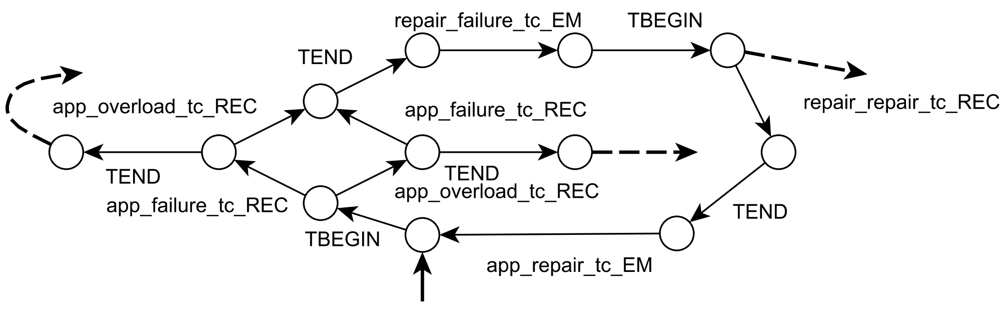


Figure 5.8 – Excerpt of the controller LTS for the running example

For this example, we used the Evaluator model checker to verify temporal properties expressed in the temporal logic formalism MCL. This has been done on a set of properties, as the two liveness properties illustrated below. The first one is checked on the controller LTS and the second one holds on the LTS of the whole system:

— The reception of a failure message by the controller is eventually followed by an emission of a repair message to the application manager in order to ask it to repair the Tomcat server

```
[true* .app_failure_tc_REC] inev (app_repair_tc_EM)
```

— The emission of an overload message by the application manager is eventually followed by an emission of a reparation or addition message by the controller

```
[true* .app_overload_tc_EM]
       inev (app_repair_tc_EM or app_add_tc_EM)
```

This property shows that the overload message is handled by the repair manager when both Tomcat failure and overload occur within a same period of time. Otherwise, it is handled by the sizing manager.

Both properties use the macro `inev (M)`, which is used especially when specifying liveness properties that are inevitability assertions. The inevitability assertions can be expressed using fixed point operators that indicate that a transition labelled with `M` eventually occurs. The inev macro is defined as follows:

```
macro inev (M) = mu X .( < true > true and [ not (M) ] X )
end macro
```

Our approach was applied for validation purposes on many illustrative examples of our dataset (managers and coordination requirements). Table 5.1 summarizes some of our experiments. Each managed application used as input is characterized using the number of managers and the coordination requirements. We give the size of the LTS (states/transitions) of the whole system as well as the controller LTS obtained after minimization (*wrt.* a strong and a weak trace relations). The last column gives the overall time to synthesize the controller.

We observe that, for some examples (gray lines), the size of the generated controller LTSs and the time required for making those LTSs grow importantly when one of the managers exhibit looping behaviours, and particularly cycles

|Managers|	Whole system LTS		Controller LTS		Time
	|states|	|transitions|	|states|	|transitions|	(m:s)
2	2,307	6,284	118	157	0:10
3	103,725	365,845	1,360	2,107	1:15
4	145	267	38	44	0:06
5	10,063,873	39,117,110	17,662	28,003	43:59
6	1,900	4,945	186	285	0:08
10	300,000	1,686,450	1,786	3,471	6:54

Table 5.1 – Experimental results: LTSs size and synthesis time

with send messages (see, *e.g.*, the 4^{th} example in Table 5.1). On a wider scale, we note that LTS sizes and generation times increase with the number of managers in parallel (see, *e.g.*, the last line of Table 5.1).

5.4.8 Deploying and Running the Generated Controller

In this section, we present some experiments we performed when deploying and running our controller for the multi-tier application introduced previously (see Fig 5.5). To do so, we used a virtualized experimental platform based on Openstack[3], which consists of six physical machines on which we instantiate virtual machines (VMs) with 1 vCPU, 2GB of memory and 8GB of disk.

The JEE multi-tier application is initially configured and deployed with a server at each tier, *i.e.*, an Apache Web server, a Tomcat server, a MySQL proxy, and a MySQL database. The initial deployment phase is automated using a dynamic management protocol allowing to connect and start the involved servers and database in the right order [7]. In a second step, we use jmeter to inject increasing load on the Apache server and thus to simulate the clients that send HTTP requests on the managed system. Once we have at least two active Tomcat servers and two MySQL databases, we start simulating failures using a failure injector. When we start injecting failures, we stop augmenting the workload on the Apache server and keep the same load for the rest of the execution.

3. https://www.openstack.org/

The failure injector is flexible and can be used for affecting any active server (Apache, Tomcat, MySQL, etc.), any number of times (single failure or multiple failures of the same or of different servers), and at any time (same period of time, different periods of time, etc.). We conducted our experiments on applications with or without the controller. We have considered various scenarios with failures of the Apache server and of the MySQL proxy as well as failures/load variation of the Tomcat servers and of the MySQL databases.

Figure 5.9 shows an excerpt of the system behaviour after 500 minutes since the application deployment. We observe that, at this moment, the application is composed of five Tomcat servers and three MySQL databases. Figure 5.9 presents several cases of failure injection. As an example, at minute 508, a failure of a replicated MySQL database causes a workload increase on the other replicated servers. These two actions happen in the same period, but the controller forwards only the failure detection to the repair manager. Accordingly, a single MySQL is added by the repair manager and the workload returns at once to its average value.

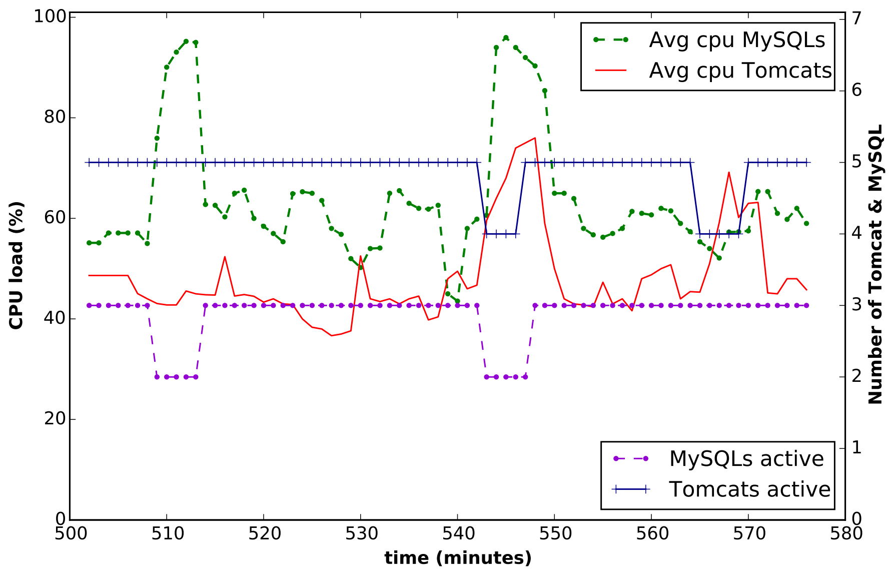


Figure 5.9 – Tomcat and MySQL failure/overload in a coordinated environment

We made several experiments in which we varied the number of failures, the Apache load, and the minimum/maximum thresholds of the Tomcat servers and of the MySQL databases. In all these cases, we observe that the controller succeeds in detecting and correcting the problems while avoiding undesired operations, that is, the unnecessary addition/removal of VMs. Figure 5.10 shows experimental results obtained with different numbers of failures. For instance, we see that when injecting 14 failures to our running application, the controller applies 18 reconfiguration operations on the system (instead of 40 without controller), and thus avoids 22 undesired operations.

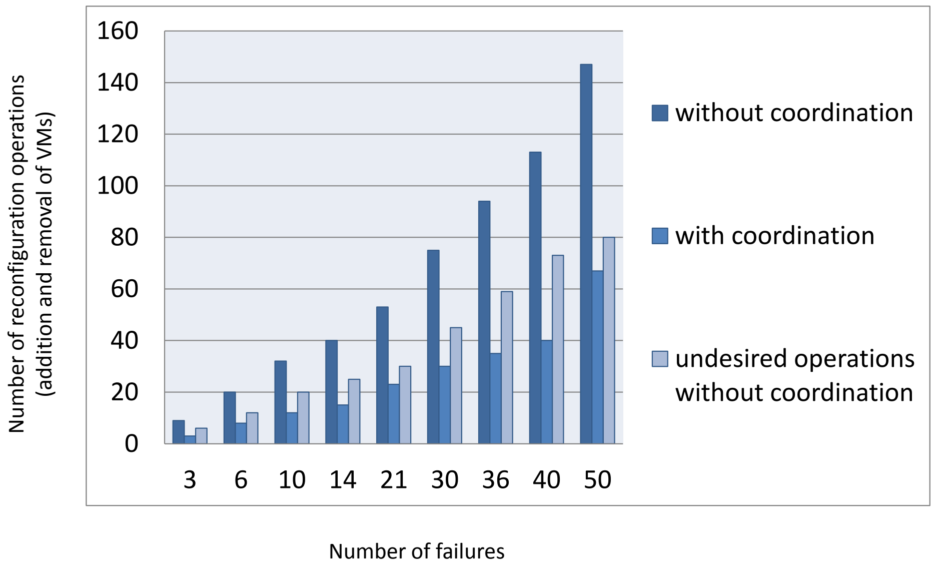


Figure 5.10 – Number of reconfiguration operations with/without coordination and number of undesired operations avoided by coordination

5.5 Conclusion

We have shown in this chapter new asynchronous synthesis techniques for generating a controller allowing to monitor and orchestrate the reconfiguration operations executed by several autonomic managers in the context of cloud applications. The generated controller interacts asynchronously with the managers, which are regarded as control loops that react to events and changes by automatically reconfiguring the application, via FIFO buffers and allows them to take globally coherent decisions in order to deal with control and incoherent decisions problems. The chapter presents also some Java code generation techniques that allows to quickly deploy the generated controller for real-world applications such as N-tier applications.
This chapter shows how the approach covers the whole development process from an expression of the requirements to the final implementation and deployment of the synthesized controller, which helps to coordinate at runtime real-world applications. In addition, these synthesis techniques can be used to control other applications where components are modelled as LTSs and communicate asynchronously. This is the case in application areas such as Web services, multi-agent systems, or hardware protocols.

6
Conclusion

" He who is not contented with what he has, would not be contented with what he would like to have. "

Socrates

Contents

THis chapter summarizes the thesis contributions and perspectives. In a first step, it concludes the major achievements of this thesis. Then, it points out, for future work, some perspectives that have been identified for the dynamic reconfiguration protocol that we designed. Some areas of prospective future research have been also identified for the improvement of the coordination requirements, which we specified for coordinating autonomic managers used for the management of distributed cloud applications.

6.1 Achievements

This Ph.D. thesis research is part of the OpenCloudware project, which is interested in building and providing an open software platform for the development of distributed applications that need to be deployed on multiple Cloud infrastructures. Within this project, we have focused on the dynamic reconfiguration of the software architecture of distributed applications at run-time and the automation of the reconfiguration tasks.
In the context of the OpenCloudware project, cloud applications are complex distributed applications that consist of interconnected software components running on several virtual machines (VMs), located on remote physical servers. The deployment and the reconfiguration of these applications are regarded as very complicated tasks, especially when one or multiple virtual machines fail when achieving them. Hence, the cloud users need protocols that allow them to dynamically reconfigure and manage their distributed applications.
We have introduced in Chapter 4 the first contribution of this thesis. We specified a novel protocol, which aims at dynamically reconfiguring distributed cloud applications and supporting multiple VM failures. The protocol enables the execution of reconfiguration operations, such as instantiating new VMs, destroying existing VMs, adding new components, and removing existing components. It also allows the resolution of dependencies among interconnected components by exchanging messages, (dis)connecting, starting, and stopping components in a specific order. The exchange of the messages between the virtual machines is ensured through a publish-subscribe communication media. Upon reception of reconfiguration operations, each machine reconfigures itself

in a decentralized manner. Indeed, each machine is equipped with a reconfiguration agent. An agent connects/disconnects and starts/stops components hosted on that machine in a defined order for preserving the application consistency. Namely, the application consistency is quite complicated due to the high parallelism degree of the protocol. We have also explained how the protocol is robust, fault-tolerant, and reconfiguration operations always terminate successfully even in the presence of a finite number of failures. We specified the protocol in a manner that it supports multiple virtual machine failures. Thus, it does not only detect VM failures but also repairs them by *(i)* creating new instances of the failed VMs and *(ii)* notifying the other VMs of this failure.
Furthermore, due to the high degree of parallelism inherent in this kind of applications, we formally specified the protocol using the LNT value-passing process algebra. We also verified it using the model checking tools available in the CADP toolbox. This toolbox turned out to be very convenient for modeling and analyzing such protocols, see [125] for a discussion about this subject. We verified about 40 properties of interest, specified using MCL, on a large number of application models and reconfiguration scenarios. More importantly, during these verification steps, the use of formal tools helped us to detect several bugs and also to improve several parts of the protocol. In particular, we deeply revised the part of the protocol dedicated to the VM destruction and component shutdown. All these detected bugs have been corrected in the corresponding Java implementation, which has been established by another partner of the OpenCloudware project.

Another contribution of this thesis was to reduce the manual execution of the reconfiguration operations, which is a major challenge for cloud applications. Autonomic computing is a promising solution for monitoring and updating automatically these kind of applications. This is achieved through the automation of administration functions and the use of autonomic managers. An autonomic manager is regarded as a control loop. It observes the applications, detects specific changes (such as the occurrence of a failure), and then reacts to them by dynamically reconfiguring the application. Multiple autonomic managers can be implemented in the same system and must make consistent decisions. Using them without coordination may lead to inconsistencies and error-prone situations. Thus, an important challenge is to use synthesis controller techniques in

order to generate a controller that aims at coordinating these managers.
We have introduced in Chapter 5 new synthesis techniques that generate a centralized controller. The particularity of these techniques is the asynchronous coordination of stateful autonomic managers through the generated controller. We have explained how the autonomic managers are modeled as LTSs. We also defined coordination requirements to describe the behavior of the controller we aim to generate. We specified these requirements using reaction rules and regular expressions. The generated controller ensures proper reconfiguration of the managed system. In fact, it coordinates the autonomic managers and communicates with them asynchronously using message passing via FIFO buffers. Our solution for controller synthesis relies on an encoding of our models and the coordination requirements into the LNT process algebra. From this encoding, an LTS can be generated using CADP compilers, hiding, and reduction techniques. This LTS exhibits all the possible executions of the controller. In this part of work, we have taken advantage of this encoding to validate the generated controller with the CADP verification tools, such as the Evaluator model checker. Indeed, since coordination requirements are written by a human being, they can be erroneous, which results in that case in an incorrect controller as well. Finally, we proposed code generation techniques to automatically obtain the Java code corresponding to the controller LTS. We validated our approach with many variants of the multi-tier Web application we presented as an example at the end of Chapter 5. Our approach covers all the development steps from the design of the coordination requirements to the actual deployment of the synthesized controller, which helps to coordinate at runtime real-world applications. We note that the generated controller can be used to control other applications where components are modeled as LTSs and communicate asynchronously. This is the case in application areas such as Web services, multi-agent systems, or hardware protocols.

6.2 Perspectives

The thesis achievements present multiple research directions for future work. This section successively describes perspectives to address the shortcomings of our contributions, addressing the dynamic reconfiguration of distributed cloud applications. Then, it presents some research directions leveraging our contributions to coordinate several autonomic managers in the cloud.

— In Chapter 4, we presented an abstraction of a model for distributed cloud applications, which is expressive enough for reconfiguration purposes. It is used for verifying the soundness of the protocol, but its primary role is to keep track of the VMs and components currently present in the application. However, it should be enhanced for other tasks.
  This model can be enriched with more details to deal with the quality of service associated with the resources. We plan to add further information to the model, such as resources availability and response time.

— In this thesis, we specified over 600 application models and reconfiguration scenarios in order to extensively validate our novel dynamic protocol. When achieving these tasks, we paid much attention to cover very different applications and scenarios, in particular pathological and corner cases. Hence, this large investment in validation makes us highly confident in the correctness of the protocol.
  This work needs to be generalized by correctness proof. To solve these gaps, proving the correctness of the protocol using theorem proving techniques would be a very interesting objective. This would complement the validation of the reconfiguration protocol using model checking techniques presented in this thesis.

— We presented in Chapter 5 a new controller synthesis techniques for coordinating multiple autonomic managers. These techniques are based on a set of reaction rules and regular expressions allowing to specify how the involved managers are supported to interact together. However, the coordination expressions do not provide the ability to specify concurrent flows (*e.g.,* by means of fork/join operators), which are quite common in orchestration and distributed coordination. Furthermore, our coordination expressions can not describe specific requirements (*e.g.,* elastic behaviors with oscillations or cool-down periods). Expressing this type of requirements require the extension of our coordination expressions and LTS model by data-awareness and real-time constraints. When numerical values, specific constraints, and requirements are added, these may, however, lead to large state-space problems, especially when addressing more complicated applications with an important number of involved managers.

Thus, some improvements are considered to integrate numerical values when describing the behavior one expects from the controller we want to generate. The enrichment of the requirements by these values allows to express specific constraints, such as a maximum/minimum permissible number of servers to a tier of replicated servers. In this view, we will also study further how to enhance the coordination expressions with more specific requirements (*e.g.,* bounds on resource usages and actuation delays) to manage complex cloud applications. Last but not least, to tackle the state space problems, we plan to integrate techniques available in the model checking, such as the Symbolic Model Checking with OBDDs and Partial Order Reduction [48].

— In this thesis, we also propose new controller synthesis techniques to generate a centralized controller for distributed cloud applications. This controller allows us to orchestrate a set of autonomic managers. Nonetheless, coordinating managers on a distributed infrastructure would rather require a distributed choreography.

The generation of distributed controllers instead of a centralized controller would be an ambitious future work. This is required in order to control the system and exchange messages without passing through a unique centralized controller, while preserving the degree of parallelism of the system.

— Last but not least, in this thesis, we use the controller LTS synthesized by our synthesis techniques to automatically generate all the Java codes for deploying it. However, for some reason there may have a need to modify the coordination requirements during the controller execution. In that case, our controller synthesis techniques should generate the new controller we want to deploy.

We will study further how to incorporate changes in the deployed controller and update them while they are running, without stopping and restarting them.

A

Appendix: LNT process grammar

```
lnt_file ::= module M [M_0, ..., M_m] is
              definition_0, ..., definition_m
             end module                            module definition
```

```
type_definition ::= type T is
                     type_expression
                    end type                       type definition
```

```
function_definition ::= function F (parameter_1, ..., ): T is
                             instructions
                        end function           function definition
```

---

process_definition ::= **process** Pi [gate_declaration$_0$, ...,]
    (parameter$_1$, ...,) **is**
        B
    end process *process definition*

---

gate_declaration ::= G$_0$, ..., G$_n$: Gamma *typed gate declaration*

| G$_0$, ..., G$_n$: **any** *polymorphic gate declaration*

---

channel_definition ::= **channel** Gamma **is**
    gate_profile$_0$, ..., gate_profile$_n$
    end channel *channel definition*

---

gate_profile ::= T$_1$, ..., T$_n$
    |(exp_declaration$_1$: T$_1$, ...,) *gate profile*

---

exp_declaration ::= X$_0$, ..., X$_n$ *experiment list*

B ::= **null** *no effect*

| **stop** *termination*

| X := E *deterministic assignment*

```
| var X1: T1, ... Xn: Tn in
    B
  end var                                  variable declaration

| B1 ; B2                                  sequential composition

| select
    B1 []       ...      [] Bn
  end select                               non-deterministic choice

| [only] if  E then B1
        [elsif Ei then Bi ... elsif Ej then Bj]
         else  Bn
    end if                                 conditional behaviour

| loop
      B
  end loop                                 forever loop

| while E loop
      B
  end loop                                 while loop

| loop L in
      B
  end loop                                 breakable loop

| case E in
    [var declaration 0 ,..., declaration n in]
      A1 -> B1
    | ...
    | An -> Bn
    | any -> Bn+1
  end case                                 case behaviour

| par [ A  in ]
```

```
      B_1  ||  ...||  B_n
end par                                       parallel composition
```

Bibliography

[1] Amazon EC2. 19

[2] AWS GovCloud (US) Region - Government Cloud Computing. 19

[3] HIPAA-Compliant Cloud Storage. 18

[4] Isabelle: Generic Proof Assistant. 26

[5] Public vs. Private Cloud Computing. 18

[6] The Coq Proof Assistant. 26

[7] R. Abid, G. Salaün, F. Bongiovanni, and N. De Palma. Verification of a Dynamic Management Protocol for Cloud Applications. In *Proc. of ATVA'13*, volume 8172 of *LNCS*, pages 178–192, Hanoi, Vietnam, 2013. Springer. 133

[8] N. Aguirre and T. Maibaum. A Logical Basis for the Specification of Reconfigurable Component-based Systems. In *Proc. of FASE'03*, FASE'03, pages 37–51, Warsaw, Poland, 2003. Springer-Verlag. 37

[9] R. Allen, R. Douence, and D. Garlan. Specifying and Analyzing Dynamic Software Architectures. In *Proc. of FASE'98*, volume 1382 of *LNCS*, pages 21–37, Lisbon, Portugal, 1998. Springer Berlin Heidelberg. 37

[10] R. Allen, R. Douence, and D. Garlan. Specifying and Analyzing Dynamic Software Architectures. In *Proc. of FASE'98*, pages 21–37, Lisbon, Portugal, 1998. 48

[11] R. Allen and D. Garlan. Formalizing Architectural Connection. In *Workshop. of ICSE'94*, ICSE '94, pages 71–80, Sorrento, Italy, 1994. IEEE Computer Society Press. 47

[12] F. Arbab. The IWIM Model for Coordination of Concurrent Activities. In *Proc. of COORDINATION '96*, COORDINATION '96, pages 34–56, Cesena, Italy, 1996. Springer-Verlag. 54

[13] F. Arbab. What Do You Mean, Coordination? In *Bulletin of the Dutch Association for Theoretical Computer Science (NVTI)*, pages 11–22, 1998. 24

[14] F. Arbab. Reo: a Channel-Based Coordination Model for Component Composition. *Mathematical Structures in Computer Science*, 14(3):329–366, 2004. 55

[15] F. Arbab, I. Herman, and P. Spilling. An Overview of Manifold and its Implementation. *Concurrency: Practice and Experience*, 5(1):23–70, 1993. 25, 54

[16] C. Attiogbé. Event-Based Approach to Modelling Dynamic Architecture: Application to Mobile Ad-Hoc Network. In *Proc. of ISoLA' 08*, pages 769–781, Kassandra, Chalkidiki, Greece, 2008. 37

[17] C. Baier and J.P. Katoen. *Principles of Model Checking*. MIT Press, 2008. 30

[18] H.M.N. Dilum Bandara and A.P. Jayasumana. Collaborative Applications over Peer-to-Peer Systems - Challenges and Solutions. *CoRR*, abs/1207.0790, 2012. 11

[19] S. Basu and T. Bultan. Automatic Verification of Interactions in Asynchronous Systems with Unbounded Buffers. In *Proc. of ASE'14*, pages 743–754, Västeras, Sweden, 2014. ACM. 119

[20] G. Behrmann, A. David, and K.G. Larsen. A Tutorial on Uppaal. In *Proc. of SFM'04*, volume 3185 of *LNCS*, pages 200–236. Springer Berlin Heidelberg, 2004. 27

[21] J. A. Bergstra and J. W. Klop. Process Algebra for Synchronous Communication. *Information and Control*, 60(1-3):109–137, 1984. 28

[22] M. Bertier, O. Marin, and P. Sens. Implementation and Performance Evaluation of an Adaptable Failure Detector. In *Proc. of DSN'02*, pages 354–363, Bethesda, Maryland, USA, 2002. IEEE Computer Society. 87

[23] K. Birman, R.V. Renesse, and W. Vogels. Adding High Availability and Autonomic Behavior to Web Services. In *Proc. of ICSE'04*, ICSE '04, pages 17–26, Scotland, UK, 2004. IEEE Computer Society. 70

[24] A. H. Bond and L. Gasser. An Analysis of Problems and Research in DAI. *Distributed Artificial Intelligence*, 1988. 24

[25] F. Boyer, O. Gruber, and D. Pous. Robust Reconfigurations of Component Assemblies. In *Proc. of ICSE'13*, ICSE '13, pages 13–22, Piscataway, NJ, USA, 2013. IEEE / ACM. 46

[26] F. Boyer, O. Gruber, and G. Salaün. Specifying and Verifying the Synergy Reconfiguration Protocol with LOTOS NT and CADP. In *Proc. of FM'11*, volume 6664 of *LNCS*, pages 103–117, Limerick, Ireland, 2011. Springer. 46

[27] D. Brand and P. Zafiropulo. On Communicating Finite-State Machines. *J. ACM*, 30(2):323–342, 1983. 119

[28] I. Brandic. Towards Self-Manageable Cloud Services. In *Proc. of COMPSAC'09*, volume 2, pages 128–133. IEEE Computer Society, July 2009. 22

[29] A. Brogi, N. Busi, M. Gabbrielli, and G. Zavattaro. Comparative Analysis of the Expressiveness of Shared Dataspace Coordination. *Electronic Notes in Theoretical Computer Science*, 62:102–115, 2001. 53

[30] A. Brogi and P. Ciancarini. The Concurrent Language, Shared Prolog. *ACM Trans. Program. Lang. Syst.*, 13(1):99–123, January 1991. 53

[31] S.D. Brookes, C.A.R. Hoare, and A.W. Roscoe. A Theory of Communicating Sequential Processes. 31(3):560–599, June 1984. 28

[32] T. Bures, P. Hnetynka, and F. Plasil. SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model. In *Proc. of SERA'06*, pages 40–48, Seattle, Washington, Aug 2006. IEEE Computer Society. 37

[33] R. Buyya, R.N. Calheiros, and X. Li. Autonomic Cloud Computing: Open Challenges and Architectural Elements. In *Proc. of EAIT'12*, pages 3–10. IEEE Computer Society, 2012. 23

[34] R. Buyya, S.Y. Chee, and S. Venugopal. Market-Oriented Cloud Computing: Vision, Hype, and Reality for Delivering IT Services as Computing Utilities. In *Proc. of CCGRID '09*, pages 5–13, Shanghai, China, Sept 2009. IEEE Computer Society. 13, 14

[35] J. Camilleri. An Operational Semantics for Occam. *International Journal of Parallel Programming*, 18(5):365–400, 1989. 28

[36] A. Cansado, C. Canal, G. Salaün, and J. Cubo. A Formal Framework for Structural Reconfiguration of Components under Behavioural Adaptation. *Electr. Notes Theor. Comput. Sci.*, 263:95–110, 2010. 37

[37] J. Cao, A. Chan, Y. Sun, and K. Zhang. Dynamic Configuration Management in a Graph-Oriented Distributed Programming Environment. *Sci. Comput. Program.*, 48(1):43–65, 2003. 11

[38] S. Carsten, K. Andreas, and K. Wolfgang. Formal Methods for the Validation of Automotive Product Configuration Data. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 17:2003, 2003. 21

[39] C.G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. 63

[40] M. Catan, R. Di Cosmo, A. Eiche, T. A Lascu, M. Lienhardt, J. Mauro, R. Treinen, S. Zacchiroli, G. Zavattaro, and J. Zwolakowski. Aeolus: Mastering the Complexity of Cloud Application Deployment. In *Proc. of ESOCC'13*, volume 8135 of *LNCS*, pages 1–3, Malaga, Spain, 2013. Springer. 44

[41] D. Champelovier, X. Clerc, H. Garavel, Y. Guerte, F. Lang, C.Mckinty, V. Powazny, W. Serwe, and G. Smeding. Reference Manual of the LOTOS NT to LOTOS Translator (Version 6.2). INRIA/VASY - INRIA/CONVECS, 2015. 32

[42] D. Champelovier, X. Clerc, H. Garavel, Y. Guerte, V. Powazny, F. Lang, W. Serwe, and G. Smeding. Reference Manual of the LOTOS NT to LOTOS Translator (Version 5.4). INRIA/VASY, 2011. 28, 93

[43] B. Chapman, M. Haines, P. Mehrota, H. Zima, and J. Van Rosendale. Opus: A Coordination Language for Multidisciplinary Applications. *Sci. Program.*, 6(4):345–362, October 1997. 53

[44] C. Chapman, W. Emmerich, F. Galán Márquez, S. Clayman, and A. Galis. Software Architecture Definition for On-demand Cloud Provisioning. In *Proc. of HPDC'10*, pages 61–72, Chicago, Illinois, 2010. ACM Press. 40

[45] C. Chapman, W. Emmerich, F.G. Marquez, S. Clayman, and A. Galis. Elastic Service Definition in Computational Clouds. In *Proc. of NOMS'10*, pages 327–334, Osaka, Japan, 2010. IEEE. 13

[46] I. Chopra and M. Singh. Analysing the Need for Autonomic Behaviour in Grid Computing. In *Proc. of ICCAE'10*, volume 1, pages 535–539, Feb 2010. 21

[47] P. Ciancarini. PoliS: a Programming Model for Multiple Tuple Spaces. In C. Ghezzi and G. Roman, editors, *workshop. of IWSSD '91*, pages 44–51, Como, Italy, 1991. IEEE Computer Society Press. 53

[48] E.M. Clarke, W. Klieber, M. Nováček, and P. Zuliani. Model Checking and the State Explosion Problem. In B. Meyer and M. Nordio, editors, *Tools for Practical Software Verification*, volume 7682 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2012. 142

[49] CloudTweaks. The 4 Primary Cloud Deployment Models, 2012. 18

[50] R.H. Cobb and H.D. Mills. Engineering Software Under Statistical Quality Control. *Software, IEEE*, 7(6):45–54, Nov 1990. 26

[51] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair. *Distributed Systems: Concepts and Design*. Addison-Wesley Publishing Company, USA, 5th edition, 2011. 12

[52] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Mine, D. Monniaux, and X. Rival. The ASTREE Analyzer. In Mooly Sagiv, editor, *Proc. of ESOP' 05*, volume 3444 of *LNCS*, pages 21–30, Edinburgh, Scotland, 2005. Springer Berlin Heidelberg. 26

[53] R. Das, J.O. Kephart, C. Lefurgy, G. Tesauro, D.W. Levine, and H. Chan. Autonomic Multi-agent Management of Power and Performance in Data Centers. In *Proc. of AAMAS '08*, pages 107–114, Estoril, Portugal, 2008. International Foundation for Autonomous Agents and Multiagent Systems. 58

[54] G. Delaval, S.M.K. Gueye, É. Rutten, and N. De Palma. Modular Coordination of Multiple Autonomic Managers. In *Proc. of CBSE'14*, pages 3–12, Lille, France, 2014. ACM. 65

[55] G. Delaval, H. Marchand, and É. Rutten. Contracts for Modular Discrete Controller Synthesis. In *Proc. of LCTES'10*, pages 57–66, Stockholm, Sweden, 2010. ACM. 64

[56] A. Eastwood. Firm Fires Shots at Legacy Systems. *Computing Canada*, 19(2):17, 1993. 12

[57] P. Emanuelsson and U. Nilsson. A Comparative Study of Industrial Static Analysis Tools. *Electron. Notes Theor. Comput. Sci.*, 217:5–21, July 2008. 26

[58] L. Erlikh. Leveraging Legacy System Dollars for E-Business. *IT Professional*, 2(3):17–23, May 2000. 12

[59] X. Etchevers, T. Coupaye, F. Boyer, and N. De. Palma. Self-Configuration of Distributed Applications in the Cloud. In *Proc. of CLOUD'11*, pages 668–675, Washington DC, USA, 2011. IEEE Computer Society. 43

[60] X. Etchevers, T. Coupaye, F. Boyer, N. De Palma, and G. Salaün. Automated Configuration of Legacy Applications in the Cloud. In *Proc. of UCC'11*, pages 170–177, Melbourne, Australia, 2011. IEEE Computer Society. 5, 43

[61] X. Etchevers, G. Salaün, F. Boyer, T. Coupaye, and N. De Palma. Reliable Self-Deployment of Cloud Applications. In *Proc. of SAC'14*, pages 1331–1338, Gyeongju, Korea, 2014. ACM Press. 43

[62] J. Fischer, R. Majumdar, and S. Esmaeilsabzali. Engage: A Deployment Management System. In *Proc. of PLDI'12*, pages 263–274, Beijing, China, 2012. ACM. 5, 41

[63] H. Garavel, F. Lang, and R. Mateescu. Compositional Verification of Asynchronous Concurrent Systems using CADP (extended version). Research Report RR-8708, INRIA Grenoble - Rhône-Alpes, Apr 2015. 28

[64] H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2011: A Toolbox for the Construction and Analysis of Distributed Processes. *International Journal on Software Tools for Technology Transfer*, 15(2):89–107, 2013. 27, 28, 94

[65] J. GEELAN. Twenty One Experts Define Cloud Computing. Virtualization, Electronic Magazine, 2008. 13, 14

[66] D. Gelernter. Generative Communication in Linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, January 1985. 25

[67] D. Gelernter and N. Carriero. Coordination Languages and Their Significance. *ACM Computing Surveys*, 35(2):97–107, February 1992. 24

[68] D. Giannakopoulou, J. Kramer, and S.C. Cheung. Behaviour Analysis of Distributed Systems Using the Tracta Approach. *Automated Software Engineering*, 6(1):7–35, January 1999. 37, 49

[69] P. Goldsack, J. Guijarro, S. Loughran, A. Coles, A. Farrell, A. Lain, P. Murray, and P. Toft. The SmartFrog Configuration Management Framework. *SIGOPS Oper. Syst. Rev.*, 43(1):16–25, 2009. 38

[70] H. Gomaa and M. Hussein. Software Reconfiguration Patterns for Dynamic Evolution of Software Architectures. In *Proc. of WICSA'04*, pages 79–88, Oslo, NORWAY, June 2004. IEEE Computer Society. 37

[71] E. Griffith. What Is Cloud Computing? , 2015. 13, 15

[72] J.F. Groote and A. Ponse. Proof Theory for μCRL: A Language for Processes with Data. In D.J. Andrews, J.F. Groote, and C.A. Middelburg, editors, *Semantics of Specification Languages (SoSL)*, Workshops in Computing, pages 232–251. Springer London, 1994. 28

[73] The Open Group. Cloud Computing Portability and Interoperability : Cloud Portability and Interoperability, 2013. 11

[74] S.M.K. Gueye, N. De Palma, É. Rutten, and A. Tchana. Coordinating Multiple Administration Loops Using Discrete Control. *SIGOPS Oper. Syst. Rev.*, 47(3):18–25, 2013. 65

[75] S.M.K. Gueye, É. Rutten, and A.Tchana. Discrete Control for the Coordination of Administration Loops. In *Proc. of UCC'12*, pages 353–358, Chicago, Illinois, USA, 2012. IEEE Computer Society. 65

[76] S. Hassan, D. Al-Jumeily, and A.J. Hussain. Autonomic Computing Paradigm to Support System's Development. In *Proc. of DESE'09*, pages 273–278. IEEE, Dec 2009. 21

[77] C.L. Heitmeyer. Formal Methods for Specifying, Validating, and Verifying Requirements. *jucs*, 13(5):607–618, May 2007. 25

[78] G.J. Holzmann. The Model Checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5):279–295, May 1997. 27

[79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, 1979. 113

[80] ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, 1989. 28, 93

[81] C. Joubert and R. Mateescu. Distributed On-the-Fly Equivalence Checking. *Electr. Notes Theor. Comput. Sci.*, 128(3):47–62, 2005. 29

[82] C. Judith et al. *NASA Formal Methods Specification and Verification Guidebook for Software and Computer Systems, Volume I: Planning and Technology Insertion*. NASA Office of Safety and Mission Assurance, Washington, DC, 1995. 25

[83] S. Kätker and K. Geihs. Fault management in QoS-enabled distributed systems. In *Proc. of DAIS'99*, pages 3–16, Ronneby, Sweden, 1999. 11

[84] J.O. Kephart. Research Challenges of Autonomic Computing. In *Proc. of ICSE'05*, pages 15–22. ACM, 2005. 22

[85] J.O. Kephart and D.M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, Jan 2003. 21, 22

[86] T. Kielmann. Designing a Coordination Model for Open systems. In P. Ciancarini and C. Hankin, editors, *Proc. of Coordination'96*, volume 1061 of *LNCS*, pages 267–284. Springer Berlin Heidelberg, 1996. 53

[87] J. Koehler, C. Giblin, D. Gantenbein, and R. Hauser. On Autonomic Computing Architectures. Technical report, IBM Zurich Research Laboratory, 2003. 22

[88] J. Kramer and J. Magee. Analysing Dynamic Change in Distributed Software Architectures. *IEE Proceedings - Software*, 145(5):146–154, 1998. 37, 49

[89] C. Krause. *Reconfigurable Component Connectors*. PhD thesis, Institute for Programming research and Algorithmics (IPA), 2011. 56

[90] C. Krause, Z. Maraikar, A. Lazovik, and F. Arbab. Modeling Dynamic Reconfigurations in Reo Using High-Level Replacement Systems. *Sci. Comput. Program.*, 76(1):23–36, 2011. 37

[91] T. A. Lascu, J. Mauro, and G. Zavattaro. Automatic Component Deployment in the Presence of Circular Dependencies. In *Proc. of FACS'13*, volume 8348 of *LNCS*, pages 254–272, Nanchang, China, 2013. 44

[92] T.A. Lascu, J. Mauro, and G. Zavattaro. A Planning Tool Supporting the Deployment of Cloud Applications. In *Proc. of ICTAI'13*, pages 213–220. IEEE Computer Society, Jun 2013. 45

[93] H. Liu, M. Parashar, and S. Hariri. A Component-Based Programming Model for Autonomic Applications. In *Proc. of ICAC '04*, pages 10–17. IEEE, May 2004. 57

[94] N. Lohmann, P. Massuthe, and K. Wolf. Operating Guidelines for Finite-State Services. In *Proc. of ICATPN' 07*, volume 4546 of *LNCS*, pages 321–341, Siedlce, Poland, 2007. Springer. 120

[95] E. Clarke M., O. Grumberg, and D.A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999. 30

[96] Greg MacSweeney. NYSE Technologies Launches Capital Markets Cloud Community Platform. 19

[97] J. Magee and J. Kramer. Dynamic Structure in Software Architectures. In *Proc. of SIGSOFT FSE'96*, pages 3–14, San Francisco, CA, USA, 1996. ACM. 37, 48

[98] J. Magee and J. Kramer. *Concurrency : State Models and Java Programs*. Wiley, Hoboken (N.J.), 2006. 37, 49

[99] J. Magee, J. Kramer, and D. Giannakopoulou. Behaviour Analysis of Software Architectures. In *Proc. of WICSA'99*, volume 12 of *IFIP Conference Proceedings*, pages 35–49, San Antonio, Texas, USA, 1999. Springer US. 37, 49

[100] T. W. Malone and K. Crowston. The Interdisciplinary Study of Coordination. *ACM Computing Surveys*, 26:87–119, 1994. 24

[101] H. Marchand, P. Bournai, M. LeBorgne, and P. Le Guernic. Synthesis of Discrete-Event Controllers based on the Signal Environment. *Discrete Event Dynamic Systems*, 10(4):325–346, 2000. 65

[102] R. Mateescu and D. Thivolle. A Model Checking Language for Concurrent Value-Passing Systems. In *Proc. of FM'08*, volume 5014 of *LNCS*, pages 148–164, Barcelona, Spain, 2008. Springer. 30, 99, 121

[103] M. Maurer, I. Breskovic, V.C. Emeakaroha, and I. Brandic. Revealing the MAPE Loop for the Autonomic Management of Cloud Infrastructures. In *Proc. of ISCC'11*, pages 147–152. IEEE Computer Society, 2011. 23

[104] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell, MA, USA, 1993. 27

[105] N. Medvidovic. ADLs and Dynamic Architecture Changes. In *Workshop. of ISAW' 96*, ISAW '96, pages 24–27, San Francisco, California, USA, 1996. ACM. 37

[106] P. Mell and T. Grance. *The NIST Definition of Cloud Computing*. National Institute of Standards and Technology, 2009. 13, 15, 16, 17, 18

[107] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer-Verlag New York, Inc., 1980. 28

[108] J. Mirkovic, T. Faber, G. Malayandisamu P. Hsieh, and R. Malavia. Dadl : Distributed application description language. Technical report, USC Information Sciences Institute and Infosys Corporation, 2010. 38

[109] J. Moad. Maintaining the Competitive Edge. *Datamation*, 4(36):61–62, 1990. 12

[110] F.A. De Oliveira, T. Ledoux, and R. Sharrock. A Framework for the Coordination of Multiple Autonomic Managers in Cloud Environments. In *Proc. of SASO'13*, pages 179–188, Philadelphia, PA, USA, 2013. IEEE. 68

[111] F.A. De Oliveira, R. Sharrock, and T. Ledoux. Synchronization of Multiple Autonomic Control Loops: Application to Cloud Computing. In *Proc. of COORDINATION '12*, volume 7274 of *LNCS*, pages 29–43, Stockholm, Sweden, 2012. Springer Berlin Heidelberg. 60

[112] N. De Palma, D. Hagimont, F. Boyer, and L. Broto. Self-Protection in a Clustered Distributed System. *IEEE Trans. Parallel Distrib. Syst.*, 23(2):330–336, 2012. 23

[113] G.K. Palshikar. Applying Formal Specifications to Real-World Software Development. *Software, IEEE*, 18(6):89–97, Nov 2001. 25

[114] L.M. Pham, A. Tchana, D. Donsez, N. De Palma, V. Zurczak, and P.Y. Gibello. Roboconf: A Hybrid Cloud Orchestrator to Deploy Complex Applications. In *Proc. of CLOUD'15*, pages 365–372, New York City, NY, 2015. IEEE. 106

[115] P. Poizat and J.C. Royer. A Formal Architectural Description Language based on Symbolic Transition Systems and Temporal Logic. *J. UCS*, 12(12):1741–1782, 2006. 37

[116] J. Prashant and D.C Schmidt. Service Configurator: A Pattern for Dynamic Configuration of Services. In *Proc. of COOTS'97*, pages 209–220, Portland, Oregon, 1997. 37

[117] R. R. Milner. *Communication and Concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989. 28

[118] G. Rackl. Multi-Layer Monitoring in Distributed Object-Environments. In *Proc. of DAIS'99*, pages 265–270. Kluwer Academic Publishers, 1999. 11

[119] P.J.G. Ramadge and W.M. Wonham. The Control of Discrete Event Systems. *Proc of the IEEE*, 77(1):81–98, 1989. 62, 63

[120] Kelton Research. New Study Shows a Staggering 4.7 Million Servers Globally Are Doing Nothing Useful, Wasting $25bn a Year, 2009. 12

[121] B. Rochwerger, D. Breitgand, E. Levy, A. Galis, K. Nagin, I. M. Llorente, R. Montero, Y. Wolfsthal, E. Elmrothand J. Cáceres, M. Ben-Yehuda, W. Emmerich, and F. Galán. The Reservoir Model and Architecture for Open Federated Cloud Computing. *IBM J. Res. Dev.*, 53(4):535–545, jul 2009. 40

[122] G. Rodosek and T. Kaiser. Determining the Availability of Distributed Applications. In A. Lazar, R. Saracco, and R. Stadler, editors, *Integrated Network Management V*, The International Federation for Information Processing, pages 207–218. Springer US, 1997. 11

[123] J. Rushby. Formal Methods and their Role in the Certification of Critical Systems. Technical Report SRI-CSL-95-1, Computer Science Laboratory, SRI International, Menlo Park, CA, Mar 1995. 25

[124] G. Salaün. Generation of Service Wrapper Protocols from Choreography Specifications. In *Proc. of SEFM'08*, pages 313–322, Cape Town, South Africa, 2008. IEEE Computer Society. 37

[125] G. Salaün, F. Boyer, T. Coupaye, N. De Palma, X. Etchevers, and O. Gruber. An Experience Report on the Verification of Autonomic Protocols in the Cloud. *ISSE*, 9(2):105–117, 2013. 139

[126] G. Salaün, X. Etchevers, N. De Palma, F. Boyer, and T. Coupaye. Verification of a Self-configuration Protocol for Distributed Applications in the Cloud. In *Proc. of SAC'12*, pages 1278–1283, Riva del Garda, Trento, Italy, 2012. ACM Press. 5, 43

[127] A. Belhaj Seboui, N. Ben Hadj-Alouane, G. Delaval, É. Rutten, and M. Yeddes. A Decentralized Supervisory Control Approach for Distributed Adaptive Systems. In *Workshop. of VECoS'10*, VECoS'10, pages 13–23, Paris, France, 2010. British Computer Society. 66

[128] A. Belhaj Seboui, N. Ben Hadj-Alouane, G. Delaval, É. Rutten, and M. Yeddes. An Approach for The Synthesis of Decentralised Supervisors for Distributed Adaptive Systems. *International Journal of Critical Computer-Based Systems*, 2(3/4):246–265, 2011. 66

[129] L.M. Vaquero, L. Rodero-Merino, and J.Caceres M. Lindner. A Break in the Clouds: Towards a Cloud Definition. *SIGCOMM Comput. Commun. Rev.*, 39(1):50–55, December 2008. 13, 15

[130] W.E. Walsh, G. Tesauro, J.O. Kephart, and R. Das. Utility Functions in Autonomic Systems. In *Proc. of ICAC'04*, pages 70–77, New York, NY, May 2004. 59

[131] L. Wang, G. Von Laszewsk, A. Younge, X. He, M. Kunze, J. Tao, and C. Fu. Cloud Computing: a Perspective Study. *New Generation Computing*, 28(2):137–146, 2010. 13, 15

[132] L. Wenjie and L. Zhanhuai. Research and Design of Autonomic Computing System Model in Cloud Computing Environment. In *Proc. of ICMT'11*, pages 5025–5028. IEEE, July 2011. 21

[133] M. Wermelinger, A. Lopes, and J. L. Fiadeiro. A Graph Based Architectural (Re)configuration Language. In *Proc. of ESEC / SIGSOFT FSE'01*, volume 26 of *ESEC/FSE-9*, pages 21–32, Vienna, Austria, 2001. ACM Press. 37

[134] T. De Wolf, G. Samaey, T.Holvoet, and D. Roose. Decentralised Autonomic Computing: Analysing Self-Organising Emergent Behaviour using

Advanced Numerical Methods. In *Proc. of ICAC'05*, pages 52–63, June 2005. 21

[135] W.M. Wonham. *Supervisory Control of Discrete-Event Systems*. Springer, 2015. 63

[136] W.M. Wonham and P.J. Ramadge. Modular Supervisory Control of Discrete-Event Systems. *Mathematics of Control, Signals and Systems*, 1(1):13–30, 1988. 64

[137] W.M. Wonham and P.J.G. Ramadge. On the Supremal Controllable Sublanguage of a Given Language. *SIAM Journal on Control and Optimization*, 25(3):637–659, 1987. 62, 63, 120

[138] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald. Formal Methods: Practice and Experience. *ACM Comput. Surv.*, 41(4):19:1–19:36, October 2009. 25

[139] P. Wyckoff, S.W. McLaughry, T.J. Lehman, and D.A. Ford. T Spaces. *IBM Syst. J.*, 37(3):454–474, July 1998. 53

[140] J. Zhou, S. Zheng, D. Jing, and H. Yang. An Approach of Creative Application Evolution on Cloud Computing Platform. In *Proc. of SAC'11*, pages 54–58. ACM, 2011. 13