Università di Pisa
Dipartimento di Informatica
Dottorato di Ricerca in Informatica
XXIX Ciclo

PhD Thesis

Modelling, analysing and reusing composite cloud applications

Candidate:
Jacopo Soldani

Supervisor:
Prof. Antonio Brogi

April 21st, 2017

Dedicated to my family

"I'm pretty sure there's a lot more to life than being really, really, ridiculously good looking. And I plan on finding out what that is."

Derek Zoolander

Abstract

How to deploy and flexibly manage complex composite applications across heterogeneous cloud platforms is one of the main concerns in enterprise IT. Vendor-agnostic models to specify the structure and management of composite cloud applications, as well as techniques for the analysis of specified applications, would permit verifying the accuracy of the design of such applications. Furthermore, the availability of techniques for reusing composite cloud applications would free developers from the need of designing and developing multiple times recurring application components.

The objective of this thesis is to propose a suitable representation for composite cloud applications, and to develop analysis and reuse techniques based upon, but not limited to, such representation. By relying on TOSCA (*Topology and Orchestration Specification for Cloud Applications*) as the reference language for syntactically describing composite cloud applications, we propose a simple modelling that permits specifying the behaviour of the management operations of an application's components. We show how their management behaviour can be modelled by *management protocols*, specified as finite state machines whose states and transitions are associated with conditions constraining the consistency of states and the executability of operations. We illustrate how to derive the management behaviour of a composite cloud application by composing the protocols of its components, and how this permits automating various analyses concerning the management of a cloud application. Last, but not least, we show how management protocols can be easily extended to take also into account the potential occurrence of faults while managing composite cloud applications.

Additionally, to enact the reuse of existing solutions, we illustrate how to syntactically matchmake (fragments of) TOSCA-based applications with respect to desired components, and how to adapt matching applications to concretely implement such components. We define two different notions of simulation between management protocols, and we exploit such notions to extend the proposed matchmaking and adaptation approaches by including the behaviour information in management protocols.

Acknowledgements

First, I would like to express my very special thanks to Antonio Brogi. Antonio has been a friendly mentor, who made me discover the world of scientific research. He taught me what scientific research is, how to carry it out, and especially how to always enjoy doing research. Without his guidance and persistent help this thesis would not have been possible.

I believe that one of the most important choices for a Ph.D. candidate is that concerning the supervisor. At the end of these three years, after all the travels we made, the many people we met, the beers, cocktails, meals and conversations we enjoyed, the meetings, endless telcos, and projects we survived to, and in general all the experiences we accumulated together and the lessons I learnt, I can only grade my initial choice with a "✓" in the so-called Brogi scale.

Special thanks are also due to Andrea Canciani, Antonio Cisternino, Filippo Bonchi, Marco Danelutto and Pengwei Wang. Andrea, Filippo and Pengwei actively cooperated with me, and their help was precious to reach some of the results in this thesis. Antonio and Marco were exposed several times to the progresses of my research work, and they were always giving me valuable feedback, especially concerning the research path to follow.

I would also like to thank Frank Leymann and Claus Pahl, who kindly hosted me as a visiting researcher in Stuttgart and in Dublin, respectively. I really appreciated working with both Frank and Claus, and I hope to have the opportunity to cooperate again with them.

I would deserve special thanks to my long-standing friends Barbara, Daniele, Federico, Federico, Flavio, Gaia, Linda, Sara, Serena, and Valerio for all the good times and relaxing moments we had together.

I would also thank all food byters for the many nerdy dinners we enjoyed together. Among all byters, special thanks are due to my friends Andrea, Francesco, Lorenzo, Marco, and Marco, with whom I also attended many classes in my previous life as a master student.

Thanks are also due to all my colleagues for all the work-break moments, and all the nice and stimulating conversations we had (which span from how to properly peel a banana, to sharing knowledge, experiences and feelings after travelling to very far places in the world). Among all colleagues, special thanks are due to Ahmad, Alessandro, and Riccardo, for the precious feedback they gave me in different occasions.

Last, but not least, a giant "thank you" is due to my father Francesco, my mother Stefania, and my brother David, who are supporting and encouraging me to pursue my dreams since I can remember, and to Sara, who is doing it since one year and a half. Without their support and constant encouragement the development of this thesis would not have been possible.

Contents

Abstract **v**

Acknowledgements **vii**

1 Introduction **1**
1.1 Research objectives 2
1.2 Research contributions 3
1.3 Thesis structure 5

2 TOSCA in a nutshell **9**
2.1 TOSCA modelling language 10
2.1.1 Topology of an application 11
2.1.2 Application components 11
2.1.3 Relations between application components 13
2.1.4 Artefacts 13
2.1.5 Management plans 14
2.1.6 Application “boundaries" 14
2.1.7 Non-functional features of an application (component) 14
2.2 Packaging TOSCA application specifications 15
2.3 Processing TOSCA application packages 15

3 Reusing cloud applications **17**
3.1 Matching cloud applications 17
3.1.1 Exact matching 18
3.1.2 Plug-in matching 20
3.1.3 Implementation 22
3.2 Overcoming naming differences 28
3.2.1 Renaming-based matching 29
3.2.2 White-box matching 32
3.2.3 Adaptation of matched cloud applications 35

4 Reusing *fragments* of application topologies **43**
4.1 A running example 43
4.2 ToscaMart 44
4.2.1 Overview 44
4.2.2 Repository of application topologies 46
4.2.3 Finding the candidate topology fragments 47

4.2.4 Election of the "best" candidate(s) 50
4.2.5 Adaptation of the elected candidate(s) 52
4.2.6 Orchestrating TOSCAMART 53
4.3 Properties of TOSCAMART 53
4.4 Implementation 58

5 Modelling and analysing cloud application management 61
5.1 Motivating scenario 62
5.2 Management protocols for cloud applications 63
5.2.1 Definition of management protocols 63
5.2.2 Characterising management protocols 65
5.3 Analysis of management protocols 66
5.3.1 Management behaviour of a composite application . 66
5.3.2 Analysing the management of a composite application 68
5.4 Implementation 70
5.5 Case study: *Thinking* 73
5.5.1 The *Thinking* application 73
5.5.2 Analysing *Thinking*'s deployment plans 77
5.5.3 Planning the undeployment of *Thinking*'s GUI and API 79
5.6 Petri net-based analysis of management protocols 80
5.6.1 Background: (Open) Petri nets 80
5.6.2 Encoding management protocols in Petri nets 80
5.6.3 Modelling the management of a cloud application . . 82
5.6.4 Analysing the management of a cloud application . . 87
5.6.5 Remarks 92

6 Behaviour-aware matching of cloud applications 95
6.1 A shortand notation for management protocols 96
6.2 Simulation-based matching 96
6.2.1 Simulation of management protocols 96
6.2.2 Behaviour-aware exact and plug-in matching 97
6.3 Flexible simulation-based matching 98
6.3.1 Flexible-simulation of management protocols 99
6.3.2 Flexible plug-in matching 101
6.4 Computing a flexible simulation 103
6.4.1 A coinductive approach to compute f-simulations . . 103
6.4.2 Properties of the approach 104

7 Fault-aware modelling and analysis of application management 111
7.1 Motivating scenario 112
7.2 Fault-aware application management protocols 113
7.2.1 Definition of fault-aware management protocols . . . 113
7.2.2 Characterising fault-aware management protocols . . 114

7.2.3 Completing fault-aware management protocols . . . 117
7.3 Analysis of fault-aware management protocols 118
7.3.1 Fault-aware management behaviour 119
7.3.2 Fault-aware analysis of application management . . . 120
7.4 Modelling and analysing "the unexpected" 122
7.5 Hard recovery 124
7.5.1 Enabling hard recovery 124
7.5.2 Planning hard recovery 128
7.6 Implementation 129
7.7 Case study: *Thinking* 134
7.7.1 Enabling fault handling and hard recovery 134
7.7.2 Planning the undeployment of *Thinking*'s GUI and API 136
7.7.3 Analysing the effects of a misbehaving component . 137

8 Related work **141**
8.1 Syntactic matching of cloud applications 141
8.2 Modelling the behaviour of cloud applications 145
8.3 Behaviour-aware matching of cloud applications 148

9 Conclusions **151**
9.1 Summary of contributions 151
9.2 Assessment of contributions 153
9.3 Possible directions for future work 156

Bibliography **159**

List of Figures

1.1 Workflow of the thesis. 6

2.1 Positioning TOSCA. 10
2.2 TOSCA service template. 11
2.3 Example of topology template. 12
2.4 Example of node type. 12
2.5 Example of relationship type. 13
2.6 Example of plan. 14

3.1 Exact matching examples. 20
3.2 Plug-in matching examples. 22
3.3 High-level management of TOSCA capabilities. 23
3.4 *ExactMatchmaker.match()* method. 24
3.5 *ExactMatchmaker.matchCapabilities()* method. 24
3.6 *PlugInMatchmaker.match()* method. 25
3.7 *PlugInMatchmaker.matchCapabilities()* method. 26
3.8 Example of node types and service templates. 27
3.9 Snapshot of the matchmaking results. 27
3.10 Pseudo-code of the adaptation of *plug-in* matched services. . 28
3.11 Renaming-based matching example. 31
3.12 Renaming-based matching example (2). 31
3.13 Function FINDOPERATIONS. 33
3.14 White-box adaptation of a *ServiceTemplate*. 35
3.15 Available service template *WebAppEnvironment*. 35
3.16 Adapting renaming-based matched service templates. 37
3.17 Target node type *WebEnv*. 37
3.18 Non-intrusive adaptation methodology: Step 1 38
3.19 Non-intrusive adaptation methodology: Steps 2 and 3 38
3.20 Non-intrusive adaptation methodology: Step 4 38
3.21 Adapting white-box matched service templates. 40
3.22 Target node type *IntegratedWebEnv*. 41
3.23 Adaptation of a white-box matched service template 41

4.1 Motivating scenario. 44
4.2 The TOSCAMART matchmaking and adaptation method. . . . 45
4.3 Examples of topologies that can be included in *Repo* 47
4.4 MATCHMAKE function. 48

4.5 MatchCaps function. 49
4.6 CandidatesUnion function. 50
4.7 Determined $Candidates$. 50
4.8 Rate function. 50
4.9 RateAndFilter function. 51
4.10 Cut function. 52
4.11 Implementation derived by ToscaMart. 53
4.12 ToscaMart function. 54
4.13 ToscaMart in the OpenTOSCA ecosystem. 58
4.14 Time performances. 59

5.1 Motivating scenario. 62
5.2 Examples of deployment plans. 63
5.3 Management protocols in our motivating scenario. 65
5.4 Well-formedness and determinism of management protocols. 66
5.5 Initial evolution according to plan (a) in Fig. 5.2. 69
5.6 Initial evolution according to plan (b) in Fig. 5.2. 69
5.7 Screenshots of Barrel. 71
5.8 A snapshot of *Thinking*. 73
5.9 Topology of the application *Thinking*. 74
5.10 Management protocol for *Mongo*'s node type. 74
5.11 Management protocol for nodes of type *Docker*. 75
5.12 Management protocol for *ThoughtsApi*'s node type. 76
5.13 Management protocol for *ThoughtsGui*'s node type. 77
5.14 Two deployment plans for *Thinking*. 77
5.15 A "fresh" instance of *Thinking*. 78
5.16 Exception raised by an instance of *ThoughtsApi*. 78
5.17 A deployment plan for *Thinking*. 79
5.18 An undeployment plan for *Thinking*. 79
5.19 Snapshot displaying the effective undeployment of *Thinking*. 79
5.20 Example of Petri net translation. 82
5.21 Example of *capability controller*. 83
5.22 Petri net encoding for the motivating scenario in Sect. 5.1. . . 84

6.1 Examples of node type and service template. 98
6.2 Adaptation of a (syntactically) matched service template. . . 98
6.3 Example of management protocols. 99
6.4 Adaptation of a flexibly plug-in matched service template. . 102

7.1 Motivating example. 112
7.2 Examples of fault-aware management protocols. 115
7.3 Example of completed management protocol. 118
7.4 Example of valid plan. 121

7.5 Example of valid sequence of operations. 121
7.6 Management protocol including unexpected behaviour. . . . 123
7.7 Example of fault injection. 123
7.8 Motivating scenario: updated topology. 125
7.9 Management protocol with hard recovery enabled. 127
7.10 Example of hard recovery plan. 129
7.11 BARREL 2.0: *Visualise* pane. 130
7.12 BARREL 2.0: *Edit* pane. 131
7.13 BARREL 2.0: *Analyse* pane. 132
7.14 Adaptation of *Thinking*'s topology enabling hard recovery. . 134
7.15 Management protocols for *DockerMongo* and *Docker*. 135
7.16 Management protocol for *ThoughtsApi*'s node type. 135
7.17 Management protocol for *ThoughtsGui*'s node type. 136
7.18 An undeployment plan for *Thinking*. 137
7.19 Snapshot displaying the effective undeployment of *Thinking*. 137
7.20 A running instance of *Thinking*. 138
7.21 Evolution of *Thinking*'s global state after a "crash". 138
7.22 Effects on *ThoughtsGui* of a crashing *ThoughtsApi*. 138
7.23 Hard recovery of *Thinking*. 140
7.24 Snapshot displaying the effective recovery of *Thinking*. . . . 140

Chapter 1

Introduction

Cloud computing has revolutionised IT, by offering a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable resources that can be rapidly provisioned and released with minimal management effort [6, 86]. Applications and resources have no more to be bought and managed on premise, but they can be simply requested (and paid) when the corresponding functionality is actually needed.

Capitalising all the benefits of cloud computing is however not that easy. Cloud applications are typically composite [55], i.e. they integrate various and heterogeneous components. The deployment, configuration, enactment, and termination of the components building up an application must hence be suitably coordinated, by also taking into account all the dependencies occurring among the application components. As the number of components grows, or the need to reconfigure becomes more frequent, application management becomes more and more time-consuming and error-prone, even if programmed by scripts or workflows [18].

Additionally, current cloud technologies suffer from a lack of standardisation, with different providers offering similar resources in a different manner [103]. As a result, to deploy and manage the same application on different cloud platforms (by fulfilling its individual requirements), application operators are often asked to design and configure most of the middleware and infrastructure layers from scratch. This requires deep technical expertise, and it results in error-prone development processes which significantly increase the costs for operating and maintaining composite cloud applications (in terms of both money and time).

> *How to deploy and flexibly manage complex composite applications over heterogeneous cloud platforms is hence a serious challenge in today's enterprise IT [82].*

This thesis aims at contributing to solve this challenge by focussing on two major issues (which have also been identified by Binz et al. [18] and by Di Cosmo et al. [47]):

(I_1) *Automate the deployment and management of composite cloud applications.* Composite cloud applications integrate various and heterogeneous

components, whose deployment, configuration, enactment, and termination must be suitably coordinated. Such coordination should be automated, as the efficient use of cloud computing peculiarities strictly depends on the degree of automation of the management of applications [82].

(I_2) *Support a vendor-agnostic design of composite cloud applications.* To achieve I_1, there is a need for vendor-agnostic, component-based models that are expressive enough to capture all the features that are typical of composite cloud applications. Such models should permit specifying the structure of a composite application by describing the components building up such application (e.g., a web-based frontend, a server, a backend database, a database management system), and the dependencies occurring among them (e.g., the frontend is deployed on the server and connected to the database, which is in turn managed by the databased management system). It is also important to capture the stateful management behaviour[1] of the components forming an application, especially to permit coordinating their management.

1.1 Research objectives

To (I_1) automate the deployment and management of composite cloud applications, by also (I_2) supporting their vendor-agnostic design, this thesis aims at advancing the state-of-the-art on:

(o_1) *Modelling composite cloud applications.* To contribute to both I_1 and I_2, there is a need for vendor-agnostic, component-based models that enables the specification of both the structure and the management of composite cloud applications.

By relying on topology graphs [20] for describing the structure of composite cloud applications, we propose a compositional modelling that permits specifying the management behaviour of application components, by also taking into account that faults eventually occur while managing complex applications [43].

(o_2) *Analysing composite cloud applications.* The availability of techniques for the analysis of composite cloud applications is also crucial to I_2, as they can permit determining whether composite cloud applications are accurately designed (e.g., checking whether the requirements of

[1]The components forming cloud applications typically have non-trivial protocols for managing them, making their contextual requirements (such as the dependencies on other components) vary over time, e.g. it may be enough to install a given component to be able to install another one, but the requirements to activate them may be different [47].

an application component are properly satisfied, or whether a workflow orchestrating the deployment of an application is valid and effectively results in deploying the application). Techniques for analysing composite cloud applications can also contribute to I_1, as they can permit automatically determining the management tasks to perform to change the actual configuration of an application (e.g., to restore the desired application configuration if a fault has happened).

In this thesis we define techniques for checking and planning the management of a composite cloud application.

(o_3) *Reusing composite cloud applications.* Cloud applications can share some management infrastructure. A concrete example is given by web applications, which can share the underlying web server needed to run them. If such server is already somehow available, it can be included in the specification of a new web application, and then suitably adapted and configured to meet its needs. Developers could indeed (I_2) describe only the application components that are specific to their solutions (e.g., those they implemented) along with their functional and non-functional requirements. Reuse techniques could then be exploited to concretise the management infrastructure needed to run them, which could then be automatically managed without any further intervention of the developer (I_1).

In this thesis we define techniques for matching and adapting existing applications, with the long-term objective of permitting to reuse them to concretely implement components of new applications.

1.2 Research contributions

We hereby list the research contributions we are going to present in this thesis. The contributions are presented under two different perspectives, by first mentioning those concerning the research objectives o_1 and o_2 (i.e., modelling and analysing composite cloud applications), and then those concertning o_3 (i.e., fostering the reuse of composite cloud applications).

Modelling and analysing composite cloud applications

A convenient way to represent complex composite applications (such as those deployed in cloud platforms) is a topology graph [20], whose nodes represent the application components, and whose arcs represent the dependencies among such components. More precisely, each topology node can be associated with the requirements of a component, the operations to manage it, and the capabilities it features. Inter-node dependencies associate the requirements of a node with capabilities featured by other nodes.

The *Topology and Orchestration Specification for Cloud Applications* (TOSCA [94]) meets this intuition, by providing a standardised modelling language for representing the topology of a cloud application. It also permits coordinating the application management, by defining (workflow) plans orchestrating the management operations offered by each component.

Unfortunately, in its current version, TOSCA does not permit specifying the behaviour of a cloud application's management operations. More precisely, it is not possible to describe the order in which the operations of a component must be invoked, nor how those operations depend on the requirements or how they affect the capabilities of that component (and hence the requirements of other components they are connected to). This implies that the verification of whether a management plan is valid can only be performed manually, with a time-consuming and error-prone process.

In this thesis we propose a extension of TOSCA that permits specifying the behaviour of the management operations of the components forming an application. We indeed show how their management behaviour can be modelled by *management protocols*, specified as finite state machines whose states and transitions are associated with conditions defining the consistency of the states of a component and constraining the executability of its management operations. Such conditions are defined on the requirements of a component, and each requirement of a component has to be fulfilled by a capability of another component. We also illustrate how to derive the management behaviour of a cloud application by composing the protocols of its components, and how this permits automating various analyses concerning the management of a cloud application, like determining whether management are valid, which are their effects, or which plans permit reaching certain application configurations.

To deal with the potential occurrence of faults (which have to be considered when managing complex cloud applications [43]), we then further extend management protocols. We indeed propose *fault-aware management protocols*, which permit modelling how nodes behave when faults occur, and we illustrate how to analyse and automate the management of composite cloud applications in a fault-resilient manner.

Notice that, even if the components of an application are described by fault-aware management protocols, the actual behaviour of such components may differ from their described behaviour (e.g., because of non-deterministic bugs [64]). We show how unexpected behaviour can be naturally modelled by automatically completing fault-aware management protocols, and how this permits analysing the (worst-possible) effects of misbehaving components on the rest of a cloud application. We also propose a way to recover composite cloud applications that are stuck because a fault was not properly handled, or because of a misbehaving component.

Fostering the reuse of composite cloud applications

To ease the design of a cloud application (e.g., a web application), we may wish to implement some of its parts by reusing already existing, validated solutions (e.g., the server stack needed to run a web application). More generally, the reuse of (fragments of) existing cloud applications can ease and speed-up the development of new applications.

In this thesis we illustrate how to reuse existing, TOSCA-based cloud applications. We first formally define how to *exact* match an application with respect to a desired application component, so as to reuse the whole application to implement such component. We then define three other types of matching (*plug-in*, *renaming-based*, and *white-box*), each permitting to identify larger sets of applications that can be adapted so as to exactly match a desired component. We also illustrate how matched applications can be adapted to exactly match a target component.

Notice that, by reusing a cloud application in its entirety, we might deploy unnecessary software (i.e., software that is not needed to concretely implement the desired application component). To tackle this issue, we further extend our matchmaking and adaptation approach by introducing and assessing TOSCAMART (*TOSCA-based Method for Adapting and Reusing application Topologies*). TOSCAMART is a method that permits reusing only the fragment of an application topology that is necessary for implementing a desired application component.

Notice that all notions of matching mentioned above are purely syntactic and do not take into account the behaviour of management operations (i.e., they do not check whether the behaviour of the operations of an available application is compatible with that of the operations of a desired application component). To overcome this limitation, we exploit the behaviour information in management protocols. Namely, we define when a desired management protocol can be *simulated* [107] by an available one, and we exploit such notion of simulation to extend the conditions constraining exact and plug-in matching. We then relax the notion of simulation into that of f-simulation (which permits simulating a desired operation with a a sequence of available operations), and we exploit f-simulation to further relax plug-in matching. We also describe a coinductive procedure to compute the function f determining an f-simulation among two management protocols (if any), and how matched applications can be adapted so as to be employed in place of desired components.

1.3 Thesis structure

The thesis is organised as illustrated in Fig. 1.1, which connects the chapters of the thesis with arrows showing which chapters directly exploit the

results presented in a chapter (e.g., the *management protocols* introduced in Chapter 5 are exploited to define the *behaviour-aware matching of cloud applications* in Chapter 6, and extended into *fault-aware management protocols* in Chapter 7). The figure also shows how chapters from 3 to 7 contribute

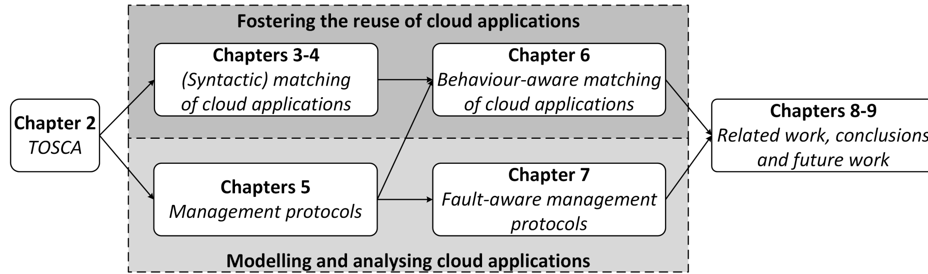


FIGURE 1.1: Workflow of the thesis.

to o_1 and o_2 (i.e., modelling and analysing cloud applications) and to o_3 (i.e., fostering the reuse of cloud applications). Further information about each chapter is given below:

In **Chapter 2** we provide a nutshell overview of TOSCA (*Topology and Orchestration Specification for Cloud Applications* [94]), by illustrating how to model complex cloud applications as service templates which orchestrate typed nodes. We also describe how TOSCA permits packaging and processing application specifications.

The nutshell introduction to TOSCA provided in Chapter 2 was published in [33], which was presented at the "3rd European Conference on Service-Oriented and Cloud Computing" (ESOCC 2014).

In **Chapter 3** we illustrate how to reuse TOSCA-based cloud applications. More precisely, after formalising four notions of (syntactic) matching between service templates and node types, we explain how matched service templates can be adapted to exactly match a target node type. We also illustrate the feasibility of our approach by means of a proof-of-concept implementation.

All notions of matching, as well as the proof-of-concept implementation and the methodology for adapting matched service templates, were published in [32], which appeared in the journal "Science of Computer Programming".

In **Chapter 4** we introduce TOSCAMART (*TOSCA-based Method for Adapting and Reusing application Topologies*), which permits reusing only the fragment of a service template that is necessary for implementing a desired node type. We also discuss the termination, soundness, and time complexity of TOSCAMART, and we illustrate its feasibility by means of a proof-of-concept implementation.

The results presented in Chapter 4 were published in [109], which appeared in the "Journal of Systems and Software".

In **Chapter 5** we propose *management protocols*, which permit modelling the management behaviour of an application and of its components. We then illustrate how such modelling permits automating various analyses of the management of complex cloud applications. We also propose an alternative, Petri-net based semantics of management protocols, and we show how various of the presented analyses can be reduced to well-known problems in the Petri nets context. Finally, to illustrate the feasibilty of our approach, we present BARREL, a web-based application that permits editing and analysing the management protocols in a TOSCA-based cloud application.

The results presented Chapter 5 were partly published in [26], which was presented at the "4th European Conference on Service-Oriented and Cloud Computing" (ESOCC 2015), and partly published in [28], which appeared in the journal "Transactions on Petri-nets and other models of Concurrency".

In **Chapter 6** we extend the syntactic matching in Chapter 3, by exploiting the behaviour information in management protocols. More precisely, we formally define the notions of simulation and f-simulation of management protocols, we exploit such notions to extend the conditions constraining exact and plug-in matching, and we illustrate how to adapt matched service templates to exactly match desired node types. We also describe a coinductive [71] procedure to compute the function f determining an f-simulation among two management protocols, and we formally assess its soundness and completeness.

The results in Chapter 6 were published in [21], which was presented at the "10th International Symposium on Theoretical Aspects of Software Engineering" (TASE 2016).

In **Chapter 7** we propose a *fault-aware* extension of *management protocols*, to permit modelling how nodes behave when faults occur, and we illustrate how to analyse and automate the management of composite cloud applications in a fault-resilient manner. We also show how fault-aware management protocols can be exploited to model and analyse the management of applications whose components are behaving unexpectedly. Finally, we present a new version of BARREL that permits editing and analysing the fault-aware management protocols in a TOSCA-based cloud application.

The results in Chapter 7 were published in [25], which was awarded as best paper at the "5th European Conference on Service-Oriented and Cloud Computing" (ESOCC 2016).

In **Chapter 8** we discuss related work, by separately treating existing solutions (i) for syntactically matching cloud applications, (ii) for modelling their management behaviour, and (iii) for matching cloud applications by taking into account their management behaviour.

In **Chapter 9** we summarise the research contributions in this thesis. We also discuss how they can contribute solving the issues I_1 and I_2, and we give perspectives for future work.

Chapter 2

TOSCA in a nutshell

TOSCA (*Topology and Orchestration Specification for Cloud Applications* [94]) is an OASIS standard whose main goal is to enable the creation of portable cloud applications and the automation of their deployment and management. In order to achieve this goal, TOSCA focuses on the following three sub-goals:

(A) *Automated application deployment and management.* TOSCA aims at providing a language to express how to automatically deploy and manage complex cloud applications [18].

This objective is achieved by requiring developers to define an abstract topology of a complex application, and to create plans describing its deployment and management [113].

(B) *Portability of application descriptions and their management.* TOSCA aims at addressing the portability of application descriptions and their management [18] (but not the actual portability of the applications themselves).

To this end, TOSCA provides a standardised way to describe the topology of multi-component applications. It also addresses management portability by relying on the portability of workflow languages used to describe deployment and management plans [19].

(C) *Interoperability and reusability of components.* TOSCA aims at describing the components of complex cloud applications in an interoperable and reusable way [18].

Interoperability is the capability for multiple components "*to interact using well-defined messages and protocols*" [94] so that they can be combined independently of the vendor(s) supplying them. TOSCA abstracts from messages and protocols details, and it permits to describe the dependencies between application components.

Furthermore, TOSCA enables defining, assembling, and packaging the building blocks of an application in a completely self-contained manner (see Sect. 2.2), thus providing a standardised way to reuse them in different applications.

Fig. 2.1 tries to position TOSCA with respect to some other standards and specifications targeting cloud interoperability[1]. More precisely, the figure positions TOSCA with respect to OASIS CAMP (*Cloud Application Management for Platforms* [92]), CIMI (*Cloud Infrastructure Management Interface* [49]), EMML (*Enterprise Mashup Markup Language* [98]), OCCI (*Open Cloud Computing Interface* [97]), Open-CSA (*Open Composite Services Architecture* [93]), OVF (*Open Virtualisation Format* [50]), SOA-ML (*Service-Oriented Architecture Modelling Language* [96]), and USDL (*Universal Service Description Language* [111]). The three sections of the pie represent the aforemen-

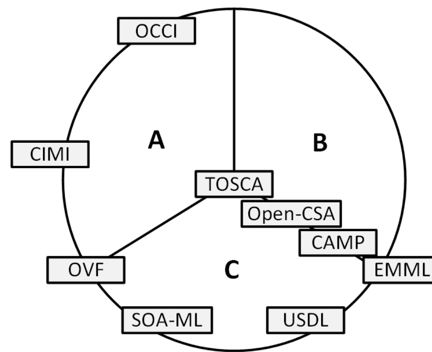


FIGURE 2.1: Positioning TOSCA with respect to other proposals of standards for cloud interoperability.

tioned three main goals of TOSCA, and the position of each label is intended to summarise "how much" the goals of an initiative overlap with the goals of TOSCA[2]. More precisely, to indicate that a standard is targeting one of the goals, its label covers the corresponding section of the pie. For instance, CAMP aims at addressing both B and C. Furthermore, if a label is not completely contained in the pie, this means that the corresponding standard only partially addresses the covered goals. Consider for instance OCCI. It provides an standardised IaaS interface which can be employed to automatise application deployment and management. Nevertheless, automation is not its real goal and thus OCCI is represented as partially covering section A and partially out of the pie.

2.1 TOSCA modelling language

To achieve the aforementioned goals, TOSCA provides an XML-based modelling language, whose purpose is to allow formalising the structure of each cloud application as a typed topology graph, and the management tasks as plans [18].

[1]A more thorough discussion on the relations between TOSCA and other cloud interoperability initiatives is provided by Pahl et al. [102].

[2]Notice that all mentioned initiatives target cloud interoperability, while only some of them also target the interoperability of application *components* (viz., C).

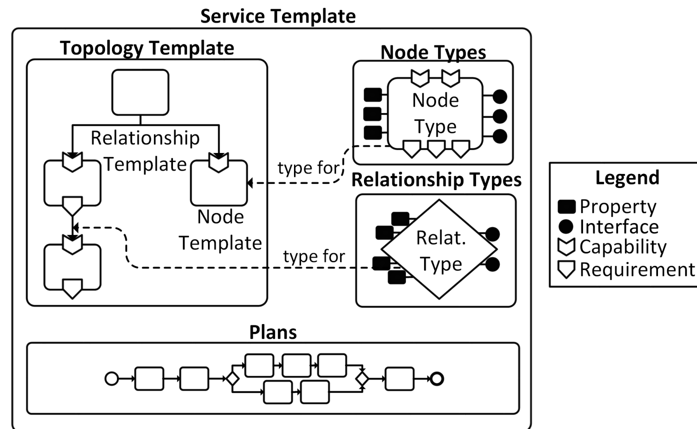


FIGURE 2.2: TOSCA service template.

An application is represented as a *service template* (Fig. 2.2), which is in turn composed by a topology template and (optionally) by some management plans. Generic type and type implementation definitions (which will be discussed later) are also contained in the XML document defining the service template as they are referred to by the templates appearing in the topology [95].

In the following we illustrate the TOSCA modelling language with reference to the *SugarCRM* application example (whose complete description can be found in the TOSCA primer [95]), which exemplifies a complex cloud application designed for enabling businesses to manage the relationships with their customer.

2.1.1 Topology of an application

The topology of a multi-component application is represented by means of a *topology template*. A topology template is essentially a typed graph whose nodes are the application components, and whose edges are the relations between such application components. Strictly speaking, the application components and their relations are represented by means of typed node templates and relationship templates, respectively. A concrete example of an application topology is shown in Fig. 2.3, which illustrates the node templates and relationship templates composing the topology of the *SugarCRM* application. Fig. 2.3 also indicates the corresponding node types and relationship types between parentheses.

2.1.2 Application components

As shown in Figs. 2.2 and 2.3, each application component appears in the topology as a *node template*, and each node template is in turn typed. This is because the purpose of node templates is to define the application-specific

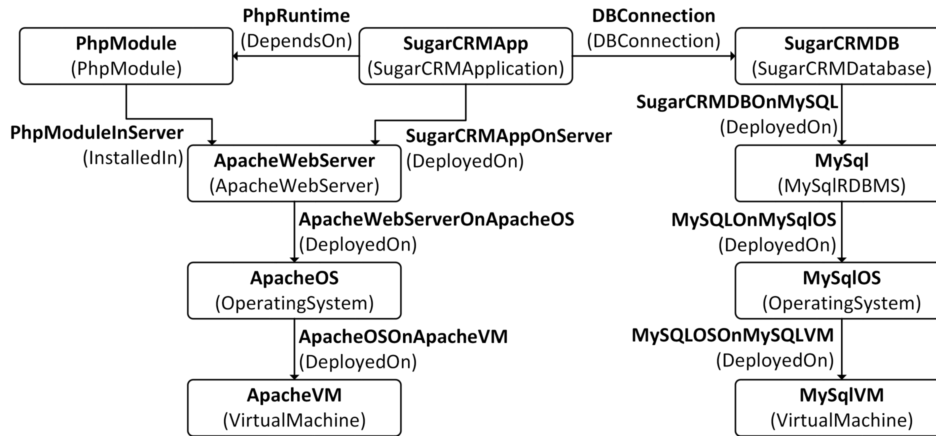


FIGURE 2.3: Example of topology template.

features of components (e.g., actual property values, QoS, etc.), while the purpose of the corresponding types is to describe the structure of the features to be specified.

The structure of the features exposed by an application component is defined by means of *node types* [18]. More precisely, a node type specifies the structure of the observable properties of an application component, the management operations it offers, the possible states of its instances, the requirements needed to properly operate it, and the capabilities it offers to satisfy other components requirements. Strictly speaking, properties are described with *property definitions*, operations with *interface* and *operation* elements, requirements with *requirement definitions* (of certain *requirement types*), and capabilities with *capability definitions* (of certain *capability types*). An example of a node type is shown in Fig. 2.4, which illustrates the struc-

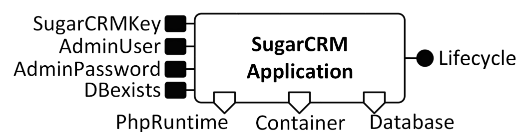


FIGURE 2.4: Example of node type.

ture of the properties, requirements and interfaces exposed by the *SugarCRMApp* component (which is of type *SugarCRMApplication* — see Fig. 2.3).

Note that node types do not specify which are the artefacts required to instantiate and operate application components, since that is the purpose of node type implementations. Each *node type implementation* refers to the node type whose implementation is under definition, and specifies its *deployment* and *implementation artefacts*. The former are the contents (viz., *artefact types* and *artefact templates*) needed to materialise instances of application components, while the latter are those which implement management operations offered by application components [19].

2.1.3 Relations between application components

Complex multi-service applications require not only to model their components, but also the relations between them [95]. As for components, relations can be modelled by means of relationship types, relationship type implementations, and relationship templates.

A *relationship type* defines the structure of a generic relationship between a *valid source* (i.e., a node type or a requirement type) and a *valid target* (i.e., a node type or a capability type). It also allows to describe the operations which can be performed on the source and on the target of the relationship (via *source interfaces* and *target interfaces*, respectively), its observable properties, and the possible states of its instances. For instance, Fig. 2.5

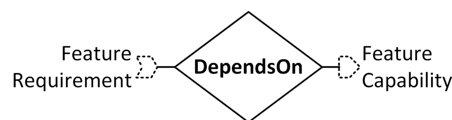


FIGURE 2.5: Example of relationship type.

illustrates the *DependsOn* relationship type, whose valid source is a *FeatureRequirement* exposed by an application component, and whose valid target is a *FeatureCapability* offered by another application component. Such a relationship type is only one of those modelling the relations between the component of the *SugarCRM* application example.

Each relationship type requires to be connected with the artefacts implementing the operations it offers. This is the purpose of *relationship type implementations*, each of which refers to a relationship type and specifies its implementation artefacts. More precisely, a relationship type implementation links each operation offered by a relationship type with the artefact types and artefact templates implementing it.

As for nodes, relationship types and type implementations only describe relations in a generic way [95]. Once placed in the topological description of a certain application, they become application-specific and thus require to be described by means of *relationship templates* (to describe application-specific features).

2.1.4 Artefacts

An artefact represents the content needed to realise the deployment or a management operation of an application component [95]. TOSCA allows artefacts to represent contents of any type (e.g., scripts, executable programs, installable images, configuration files, libraries, etc.). This permits describing artefacts along with the metadata needed to properly process them. The structure of such metadata is described by means of *artefact types*,

while links to concrete artefacts (and values of invariant metadata) can be specified by employing *artefact templates*.

2.1.5 Management plans

Plans enable the description of deployment and management aspects of an application [77]. Each *plan* is a workflow combining the management operations offered by the nodes in the application topology. TOSCA prescribes to use workflows to describe plans (so as to leverage of their suitability to handle errors, exceptions and human interactions [19]), but it does not mandate the use of a specific workflow language [18]. Furthermore, plans are distinguished on the basis of their type. There are only two predefined types of plans: The *build plan* type models plans which initially create a new instance of a service template, while the *termination plan* type is for plans used to terminate the existence of a service template instance.

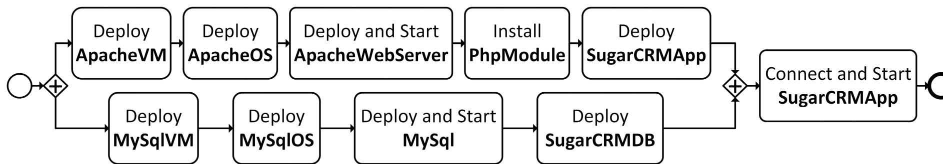


FIGURE 2.6: Example of plan.

A concrete example of a TOSCA plan is shown in Fig. 2.6, which illustrates a possible (BPMN) build plan for the *SugarCRM* application example.

2.1.6 Application “boundaries"

A service template can also describe the functional and non-functional features it exposes externally. The (optional) boundary definitions permit specifying the properties, capabilities, requirements and operations of internal components which are externally visible. It also allows to expose management plans as operations and to describe the non-functional properties of the complex application.

2.1.7 Non-functional features of an application (component)

TOSCA employs policies to describe non-functional behaviour, or the QoS (Quality-of-Service) that an application and its components can declare to expose [18]. Similar to the other entities in the TOSCA standard, a policy has an abstract *policy type* definition and is instantiated by defining a *policy template*. While the policy type describes the structure and required parameters of a policy, the policy template is used to define a specific policy instance.

Service templates (via boundary definitions), node templates, and relationship templates can then declare their non-functional features by referring the policy types or policy templates describing them [112].

2.2 Packaging TOSCA application specifications

TOSCA prescribes the format to archive application specifications along with the installable and executable files needed to properly instantiate the specified applications. This is because the modelling language illustrated in the previous section only allows developers to specify the application topology and its management in a *.tosca* document. Such document must be packaged together with the artefacts implementing its components so as to make them available to the execution environment.

The TOSCA specification defines an archive format called CSAR (*Cloud Serivce ARchive*) to package application specification together with concrete implementation and deployment artifacts. A CSAR is a (compressed) zip file containing at least the *definitions* and *TOSCA metadata* directories.

The *definitions* directory contains one or more **.tosca* documents. These documents contain the TOSCA definitions describing the cloud application. More precisely, exactly one of them must contain the service template defining the structure and behaviour of the whole cloud application, while the others can be devoted to supporting definitions (so as to modularise the application specification). Additionally, CSARs can also be devoted to contain TOSCA definitions to be reused in other contexts. For instance, a CSAR might be used to provide a set of node types (with their corresponding implementations) to be employed as building blocks while specifying new cloud applications.

A *TOSCA metadata* directory contains the *TOSCA.meta* file. Its purpose is to describe metadata about the other files in the CSAR by means of blocks, which in turn consist of a set of name-value pairs. The first block of the *TOSCA.meta* file provides metadata about the CSAR itself (e.g., version, creator, etc.), while each other block points to a file in the CSAR and describes its metadata.

2.3 Processing TOSCA application packages

As we just explained, an application specification is packaged (along with the concrete artefacts implementing its components) in a CSAR archive with the purpose of deploying it on cloud platforms. Subsequently, a cloud platform is TOSCA-compliant if it offers a TOSCA container (e.g., OpenTOSCA [15]) which is an engine able to process CSAR archives, and thus to deploy and operate the applications they contain.

TOSCA containers can deploy applications by processing the CSAR archives in two different ways [95]. On one hand, *imperative processing* takes the CSAR and deploys the application according to the workflow defined as a build plan in the corresponding service template (e.g., the build plan shown in Fig. 2.6). On the other hand, *declarative processing* deploys the application by trying to automatically excerpt a deployment plan from its topology template. In the latter case, the CSAR engine (a) first deploys the nodes of an application without requirements on other nodes, and then (b) until all nodes have been deployed, it searches the nodes whose requirements are satisfied (by the capabilities of the already deployed nodes) and deploys them. For instance, if we consider the topology in Fig. 2.3, the declarative processing works as follows. First, it deploys the node templates *ApacheVM* and *MySqlVM* since they have no dependencies on other nodes. Second, it deploys *ApacheOS* and *MySqlOS* since the node templates they depend on have been deployed. Then, it proceeds in repeating steps analogous to the second one until all the node templates in the topology have been deployed.

TOSCA containers not only have to support application at deployment time, but also at run time. They are indeed in charge of ensuring that the implementation artefacts (corresponding to management operations) are available [77]. They should also be able to properly operate such artefacts as well as the management plans provided by the application specification [18].

Chapter 3

Reusing cloud applications

According to the TOSCA primer [95], a node type can be made concrete by substituting it by a service template, provided that the latter exposes the same features as the former on its boundaries. While such matching is mentioned in the primer with reference to an example, no formal definition of matching is given either in TOSCA or in the primer.

In Sect. 3.1 we first formalise the notion of *exact* matching ($\equiv$) between service templates and node types mentioned in the TOSCA primer [95]. We then define the notion of *plug-in* matching ($\simeq$), which relaxes the exact one and identifies larger sets of service templates that can be adapted so as to (exactly) match a node type. We also illustrate the feasibility of the proposed notions of matching by means of a proof-of-concept implementation.

In Sect. 3.2 we define two other types of matching, called *renaming-based* matching ($\sim$) and *white-box* matching ($\Box$), each permitting to ignore larger sets of naming differences when matching service templates with respect to node types. While exact and plug-in matching are purely syntactic, both renaming-based and white-box matching require a notion of semantic equivalence to permit ignoring naming differences. To avoid all semantics-related issues (e.g., employing ontology-based descriptions of cloud services, and cross-ontology matchmaking [75, 85]), in Sect. 3.2 we also propose a methodology to manually adapt renaming-based and white-box matching service templates. More precisely, we show how to exactly match a target node type by non-intrusively adapting a renaming-based matched service template, or by intrusively adapting a white-box matched service template.

3.1 Matching cloud applications

In this section we first formally define when a service template can *exactly* match ($\equiv$) a node type. Then, we formally define the *plug-in* matching ($\simeq$), which relaxes the exact one (viz., $\equiv\subset\simeq$) in order to identify larger sets of service templates that can be adapted so as to (exactly) match a node type. Finally, we show a proof-of-concept implementation of the introduced notions of matching.

3.1.1 Exact matching

We hereby formalise the definition of *exact* matching between a service template S and a node type N, which mirrors the informal definition of matching mentioned in the TOSCA primer [95]. The following definition specifies when S exactly matches N in terms of the requirements (Reqs), capabilities (Caps), policies (Pols), properties (Props) and interfaces (Ints) of S and N[1].

Definition 3.1 (Exact matching). *Let N be a node type and let S be a service template. S* exactly *matches N ($S \equiv N$) iff:*

$$\begin{array}{lrcll} (1) & \mathsf{Reqs}(S) & \equiv_R & \mathsf{Reqs}(N) & \textit{and} \\ (2) & \mathsf{Caps}(S) & \equiv_C & \mathsf{Caps}(N) & \textit{and} \\ (3) & \mathsf{Pols}(S) & \equiv_{PO} & \mathsf{Pols}(N) & \textit{and} \\ (4) & \mathsf{Props}(S) & \equiv_{PR} & \mathsf{Props}(N) & \textit{and} \\ (5) & \mathsf{Ints}(S) & \equiv_I & \mathsf{Ints}(N). & \end{array}$$

Before digging into the details of conditions (1—5), we introduce some shorthand notations to retrieve names and types of TOSCA elements.

Notation 3.1. *Let N be a node type and let S be a service template. Then:*

- $\mathsf{name}(x)$ *denotes the* name *of x, where x can be a requirement, capability, property, interface, operation, or parameter of N or S.*
- $\mathsf{type}(x)$ *denotes the* type *of x, where x can be a requirement, capability, policy, property, or parameter of N or S.*
- $\mathsf{XMLtype}(x)$ *denotes the* XML type *of x, where x can be a property (or a set of properties) of N or S.*

We now define the *exact* matching of requirements. Essentially, they must have the same name and type, and they must be in a one-to-one correspondence. The same holds for capabilities.

Definition 3.2 (Exact matching of requirements and capabilities). *Let N be a node type and let S be a service template. Then:*

$\mathsf{Reqs}(S) \equiv_R \mathsf{Reqs}(N)$ *iff*

$$\forall r_S \in \mathsf{Reqs}(S)\ \exists! r_N \in \mathsf{Reqs}(N) : \mathsf{name}(r_S) = \mathsf{name}(r_N) \wedge \mathsf{type}(r_S) = \mathsf{type}(r_N)$$

and

$$\forall r_N \in \mathsf{Reqs}(N)\ \exists! r_S \in \mathsf{Reqs}(S) : \mathsf{name}(r_N) = \mathsf{name}(r_S) \wedge \mathsf{type}(r_N) = \mathsf{type}(r_S).$$

$\mathsf{Caps}(S) \equiv_C \mathsf{Caps}(N)$ *iff*

[1] Strictly speaking, the definition relates the requirements exposed by S with the requirement definitions of N, the capabilities exposed by S with the capability definitions of N, the policies exposed by S with the policy types applicable to N, and the properties exposed by S with the properties definition declared by N.

$$\forall c_S \in \mathsf{Caps}(S)\ \exists! c_N \in \mathsf{Caps}(N) : \mathsf{name}(c_S) = \mathsf{name}(c_N) \wedge \mathsf{type}(c_S) = \mathsf{type}(c_N)$$

and

$$\forall c_N \in \mathsf{Caps}(N)\ \exists! c_S \in \mathsf{Caps}(S) : \mathsf{name}(c_N) = \mathsf{name}(c_S) \wedge \mathsf{type}(c_N) = \mathsf{type}(c_S).$$

According to the TOSCA specification [94], a policy type can be associated with a set of node types to which it is applicable[2]. To ensure exact matching, the type of each policy of S must therefore be one of the policy types applicable to N.

Definition 3.3 (Exact matching of policies)**.** *Let N be a node type and let S be a service template. Then:*

$$\mathsf{Pols}(S) \equiv_{PO} \mathsf{Pols}(N) \textit{ iff } \forall pol_S \in \mathsf{Pols}(S) : \mathsf{type}(pol_S) \in \mathsf{Pols}(N).$$

Furthermore, since a node type only specifies the XML schema of its observable properties (while service templates specify actual values of properties), property matching reduces to comparing XML types.

Definition 3.4 (Exact matching of properties)**.** *Let N be a node type and let S be a service template. Then:*

$$\mathsf{Props}(S) \equiv_{PR} \mathsf{Props}(N) \textit{ iff } \mathsf{XMLtype}(\mathsf{Props}(S)) = \mathsf{XMLtype}(\mathsf{Props}(N)).$$

Finally, interfaces must have the same name and must be in a one-to-one correspondence. The same holds for interface operations and for operation parameters. Operation parameters must also have the same type.

Definition 3.5 (Exact matching of interfaces)**.** *Let N be a node type and let S be a service template. Then:*

$\mathsf{Ints}(S) \equiv_I \mathsf{Ints}(N)$ *iff*

$$\forall i_S \in \mathsf{Ints}(S)\ \exists! i_N \in \mathsf{Ints}(N) : \mathsf{name}(i_S) = \mathsf{name}(i_N) \wedge \forall o_S \in \mathsf{Ops}(i_S)\ \exists! o_N \in \mathsf{Ops}(i_N) : o_S \equiv_o o_N$$

and

$$\forall i_N \in \mathsf{Ints}(N)\ \exists! i_S \in \mathsf{Ints}(S) : \mathsf{name}(i_N) = \mathsf{name}(i_S) \wedge \forall o_N \in \mathsf{Ops}(i_N)\ \exists! o_S \in \mathsf{Ops}(i_S) : o_N \equiv_o o_S$$

where $\mathsf{Ops}(.)$ *denotes the set of operations of an interface and where*

$o_x \equiv_o o_y$ *iff*

$\mathsf{name}(o_x) = \mathsf{name}(o_y)$ *and*

$|\mathsf{I}(o_x)| = |\mathsf{I}(o_y)|$ *and*

$|\mathsf{O}(o_x)| = |\mathsf{O}(o_y)|$ *and*

$\forall a \in \mathsf{I}(o_x), \exists! b \in \mathsf{I}(o_y) : \mathsf{name}(a) = \mathsf{name}(b) \wedge \mathsf{type}(a) = \mathsf{type}(b)$*, and*

$\forall a \in \mathsf{O}(o_x), \exists! b \in \mathsf{O}(o_y) : \mathsf{name}(a) = \mathsf{name}(b) \wedge \mathsf{type}(a) = \mathsf{type}(b)$*, and*

where $\mathsf{I}(o)$ *and* $\mathsf{O}(o)$ *denote the input and output parameters of operation o.*

[2] We assume that a policy type is applicable to all node types if not specified otherwise.

It is easy to observe that the notion of exact matching is quite strict, as illustrated by the following example.

Example 3.1. Consider the node types N_1 and N_2 and the service template S of Fig. 3.1, where C and C_{sup} denote sets of capabilities, R and R_{sub} denote sets of requirements, p_j denotes a property, i_j denotes an interface, o_j denotes an operation, and where policies and operation parameters are omitted for readability. Suppose

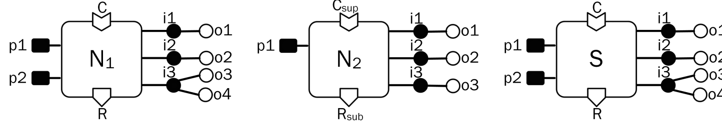


FIGURE 3.1: Exact matching examples.

that S exactly matches N_1 (viz., $S \equiv N_1$) and that N_2 differs from N_1 since it exposes "more" requirements than N_1 and "less" capabilities, properties and operations than N_1. While, according to Defs. 3.1—3.5, S does not exactly match N_2 (viz., $S \not\equiv N_2$), a less strict definition of matching should allow S to match also N_2 (as we will discuss in the next section). □

3.1.2 Plug-in matching

Intuitively speaking, a service template plug-in matches a node type if the former "requires less" and "offers more" than the latter. Analogously to Def. 3.1, the following definition specifies when a service template S can plug-in match a node type N in terms of the requirements, capabilities, policies, properties and interfaces of S and N. As node types do not specify concrete policies (just applicable policies), the matching of policies ($\equiv_{PO}$) is unchanged.

Definition 3.6 (Plug-in matching). *A service template S* plug-in *matches a* NodeType *N ($S \simeq N$) iff:*

(1) $\mathsf{Reqs}(S) \simeq_R \mathsf{Reqs}(N)$ *and*
(2) $\mathsf{Caps}(S) \simeq_C \mathsf{Caps}(N)$ *and*
(3) $\mathsf{Pols}(S) \equiv_{PO} \mathsf{Pols}(N)$ *and*
(4) $\mathsf{Props}(S) \simeq_{PR} \mathsf{Props}(N)$ *and*
(5) $\mathsf{Ints}(S) \simeq_I \mathsf{Ints}(N)$.

Intuitively speaking, a service template has to expose "less" requirements than a node type. According to TOSCA [94], names of requirements cannot be different, but types do not need to strictly coincide (viz., the type of each requirement of the node type can be a subtype of the type of the homonym requirement in the service template).

Notation 3.2. *We write $t' \geq t$ when type t' extends*[3] *or is equal to type t.*

[3]More precisely, if t and t' are TOSCA elements then t' extends t if t' is (directly or undirectly) *derived from* t. If t and t' are instead XML types then the standard notion of XML extension applies.

Definition 3.7 (Plug-in matching of requirements)**.** *Let N be a node type and let S be a service template. Then:*

$$\begin{aligned}&\mathsf{Reqs}(S) \simeq_R \mathsf{Reqs}(N) \textit{ iff}\\&\quad \forall r_S \in \mathsf{Reqs}(S)\ \exists r_N \in \mathsf{Reqs}(N) : \mathsf{name}(r_N) = \mathsf{name}(r_S) \wedge\\&\quad\quad \mathsf{type}(r_N) \geq \mathsf{type}(r_S).\end{aligned}$$

Dually, a service template must expose "more" capabilities and properties of a node type. According to the TOSCA specification [94], names of capabilities cannot be different, but types do not need to strictly coincide (viz., the type of each capability of the service template can be a subtype of the type of the homonym capability in the node type).

Definition 3.8 (Plug-in matching of capabilities and properties)**.** *Let N be a node type and let S be a service template. Then:*

$$\begin{aligned}&\mathsf{Caps}(S) \simeq_C \mathsf{Caps}(N) \textit{ iff}\\&\quad \forall c_N \in \mathsf{Caps}(N)\ \exists c_S \in \mathsf{Caps}(S) : \mathsf{name}(c_S) = \mathsf{name}(c_N) \wedge\\&\quad\quad \mathsf{type}(c_S) \geq \mathsf{type}(c_N).\\&\mathsf{Props}(S) \simeq_{PR} \mathsf{Props}(N) \textit{ iff}\\&\quad \mathsf{XMLtype}(\mathsf{Props}(S)) \geq \mathsf{XMLtype}(\mathsf{Props}(N)).\end{aligned}$$

Finally, a service template must expose all the operations exposed by a node type. The matching can focus on operations and abstract from (names of) interfaces.

Definition 3.9 (Plug-in matching of interfaces)**.** *Let N be a node type and let S be a service template. Then:*

$$\begin{aligned}&\mathsf{Ints}(S) \simeq_I \mathsf{Ints}(N) \textit{ iff}\\&\quad \forall i_N \in \mathsf{Ints}(N), o_N \in \mathsf{Ops}(i_N) : \exists i_S \in \mathsf{Ints}(S), o_S \in \mathsf{Ops}(i_S) :\ o_S \equiv_o o_N.\end{aligned}$$

It is worth noting that when a service template S plug-in matches a node type then S can be easily adapted into a new service template S' that exactly matches that node type. Such S' is built by creating a new service template having S as its only node, and by simply exposing (via the boundary definitions of S') the capabilities, policies, properties, and interfaces of the node type to be matched. If requirements plug-in match (but do not exactly match) then a dummy *NoBe* node template is introduced to artificially extend the set of requirements of S so as to expose the same requirements of the node type to be matched.

Example 3.2. Example 3.1 illustrated a service template S that does not exactly match a node type N_2 since the latter exposes "more" requirements and "less" capabilities, properties and operations than the former. Since S exposes one property ($p2$) and one operation ($o4$) more than N_2, we have that $\mathsf{Props}(S) \simeq_{PR} \mathsf{Props}(N_2)$ and $\mathsf{Ints}(S) \simeq_{PR} \mathsf{Ints}(N_2)$ by Defs. 3.8 and 3.9, respectively. Therefore, if $R \simeq_R R_{sub}$

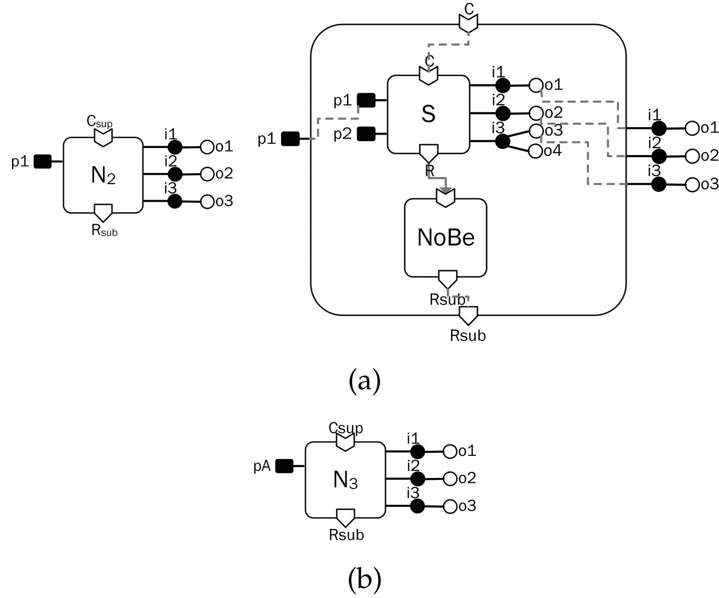


FIGURE 3.2: Plug-in matching examples.

and $C \simeq_R C_{sup}$ hold too, then S plug-in matches N_2 ($S \simeq N_2$). Fig. 3.2.(a) illustrates how S can be adapted to exactly match N_2.

Consider now the node type N_3 in Fig. 3.2.(b), which differs from N_2 only since it exposes property pA instead of property $p1$. According to Def. 3.8, S does not plug-in match N_3 ($S \not\simeq N_3$). However, if $p1$ and pA were (syntactically) different names for the same property and if the type of $p1$ were compatible with the type of pA (i.e., $\texttt{type}(p1) \geq \texttt{type}(pA)$), then a less strict definition of matching should allow S to match also N_3. □

3.1.3 Implementation

The definitions of matching presented in Sects. 3.1.1 and 3.1.2 can be fruitfully integrated in the OpenTOSCA open source environment [15], as well as in other TOSCA implementations currently under development, in order to enhance their type-checking capabilities. Since OpenTOSCA is written in Java, we shall now describe a proof-of-concept Java implementation[4] of both *exact* and *plug-in* matchings.

High-level modelling of TOSCA

The OpenTOSCA environment directly exploits TOSCA XSD[5] to automatically generate the Java representation of TOSCA files. The obtained representation is quite low-level and it does not ease the development of integrated plug-ins. For example, consider the management of the relationships between capabilities, capability types and capability definitions. In

[4] The source code is publicly available on GitHub at `https://github.com/jacopogiallo/Finding-available-services-in-TOSCA-compliant-clouds`.

[5] `http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/schemas/TOSCA-v1.0.xsd`.

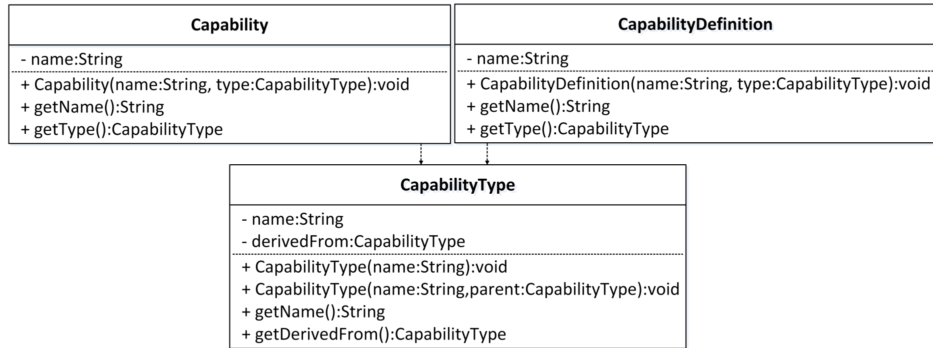


FIGURE 3.3: High-level management of TOSCA capabilities.

TOSCA both capabilities and capability definitions can reference capability types by means of qualified names. To avoid that the automated XSD-based conversion of TOSCA capabilities and capability definitions looses such references, ad-hoc mechanisms must be developed to generate an explicit representation of such references that associates qualified names with the corresponding Java classes.

Since OpenTOSCA and Winery do not provide a high-level API to manage TOSCA elements [13], we will employ a higher level Java representation of TOSCA elements, which is still a hierarchy of classes that corresponds to the hierarchy of elements defined in the TOSCA XSD. For instance, the management of capabilities, capability types and capability definitions is performed by directly referencing the corresponding Java objects (Fig. 3.3). Thanks to its schema definition orientedness, such higher level representation can be easily mapped on the lower level representations currently employed by the available TOSCA implementations[6].

Implementation of the matchmakers

Since *plug-in* matching generalises *exact* matching (viz., $\equiv \subset \simeq$), we implemented the two matchings as a class hierarchy. The top element of such hierarchy is the abstract *Matchmaker* class. It groups the fields and methods common to both the *exact* and the *plug-in* matchmakers. More precisely, it declares the service template s and the node type n to be matched, and the sets of unmatched elements (e.g., *unmatchedCapabilities*). It also provides the constructor method, as well as the abstract methods to check whether s *match*es n and to access the above mentioned sets of unmatched elements (e.g., *getUnmatchedCapabilities*).

The abstract *Matchmaker* class is then extended to provide the implementation of the *exact* matchmaker. The resulting *ExactMatchmaker* suitably stores the exactly matched TOSCA elements (e.g., *exactlyMatchedCapabilities*) and

[6]The documentation of the higher level API is available at http://jacopogiallo.github.io/Finding-available-services-in-TOSCA-compliant-clouds/.

```
1 public boolean match() {
2   matchCapabilities();
3   matchRequirements();
4   matchPolicies();
5   matchProperties();
6   matchInterfaces();
7   return (areCapabilitiesMatched && areRequirementsMatched &&
8     arePoliciesMatched && arePropertiesMatched && areInterfacesMatched);
9 }
```

FIGURE 3.4: *ExactMatchmaker.match()* method.

```
1  protected void matchCapabilities() {
2    exactlyMatchedCapabilities = new ArrayList<Capability>();
3    areCapabilitiesMatched = false;
4    unmatchedCapabilities = n.getCapabilityDefinitions().getList();
5    List<Capability> sCaps =
6      s.getBoundaryDefinitions().getCapabilities().getList();
7    List<CapabilityDefinition> newUnmatchedCapabilities =
8      new ArrayList<CapabilityDefinition>();
9    boolean matched;
10   for(CapabilityDefinition cDef : unmatchedCapabilities) {
11     matched = false;
12     for(Capability c : sCaps) {
13       matched = match(cDef, c);
14       if(matched) {
15         exactlyMatchedCapabilities.add(c);
16         break;
17       }
18     }
19     if(!matched) newUnmatchedCapabilities.add(cDef);
20   }
21   unmatchedCapabilities = newUnmatchedCapabilities;
22   if(n.getCapabilityDefinitions().getList().size() != sCaps.size())
23     return;
24   if(unmatchedCapabilities.isEmpty())
25     areCapabilitiesMatched = true;
26   }
27
28   protected boolean match(CapabilityDefinition cDef, Capability c) {
29     return (cDef.getName().equals(c.getName()) &&
30       cDef.getCapabilityType().getName().equals(c.getType().getName()));
31   }
```

FIGURE 3.5: *ExactMatchmaker.matchCapabilities()* method.

provides access to them (e.g., *getExactlyMatchedCapabilities*). It also implements the *match* method, which checks whether a service template *s exactly* matches a node type *n* (Fig. 3.4). The matching is performed in a step-wise way, to keep it aligned with Def. 3.1 (lines 2-6). Each kind of element is matched with a separate method (e.g., *matchCapabilities*) which properly instantiates the corresponding boolean variable (e.g., *areCapabilitiesMatched*). The result of the whole matchmaking is given by the logical *and* among all sub-results (lines 7-8).

Consider, for instance, the matchmaking of capabilities[7] (Fig. 3.5). After initialization (lines 2-9), the method checks whether all the capabilities defined by the node type *n* are present on the boundaries of *s*. More precisely, for each capability definition in *n* (line 10) it checks whether there exists a

[7]The (exact) matchmaking of the other TOSCA elements is analogous.

```
public boolean match() {
  super.matchCapabilities();
  super.matchRequirements();
  super.matchPolicies();
  super.matchProperties();
  super.matchInterfaces();
  if(areCapabilitiesMatched && areRequirementsMatched &&
    arePoliciesMatched && arePropertiesMatched && areInterfacesMatched)
      return true;

  if(!areCapabilitiesMatched) matchCapabilities();
  if(!areRequirementsMatched) matchRequirements();
  if(!arePropertiesMatched) matchProperties();
  if(!areInterfacesMatched) matchInterfaces();
  return (areCapabilitiesMatched && areRequirementsMatched &&
    arePoliciesMatched && arePropertiesMatched && areInterfacesMatched);
}
```

FIGURE 3.6: *PlugInMatchmaker.match()* method.

capability on the boundaries of s such that they exactly match (lines 11-18). The comparison is performed by the *match* method which checks whether a capability definition and a capability have same name and type (lines 28-31). If no capability matches the capability definition under consideration, then the latter is added to a new set of unmatched capability definitions (line 19). After the end of the loop, the set of *unmatchedCapabilities* is updated (line 21). Then, to ensure the one-to-one correspondence needed by Def. 3.2, the method checks whether both n and s expose the same number of capabilities (lines 22-23). If so, and if there are no unmatched capability definitions, then the *areCapabilitiesMatched* variable is set to *true* (lines 24-25). Otherwise, the method ends (by leaving it set to *false*).

The *ExactMatchmaker* is in turn extended by the *PlugInMatchmaker*. The latter stores and provides access to the plug-in matched TOSCA elements (e.g., via the field *plugInMatchedCapabilities* and through the method *getPlugInMatchedCapabilities*) and overrides the *match* method by making it check whether a node type n *plug-in* matches a service template s (Fig. 3.6). The method starts by checking whether the two elements exactly match (lines 2-9). If this is not the case, the plug-in matching of (unmatched) TOSCA elements is performed separately (lines 11-14). Finally, the whole matchmaking result is computed with the logical *and* among all partial results (lines 15-16).

Consider, for instance, the matchmaking of capabilities[8] (Fig. 3.7). Since it is performed after the exact matching, the set up of the environment is lighter than that of Fig. 3.5 (lines 2-7). The method then proceeds by checking whether all the capabilities defined by n are compatible with those on the boundaries of s. More precisely, for each capability definition in n (that has not yet been matched — line 8), it checks whether there exists a capability on the boundaries of s such that they plug-in match (lines 9-16). The

[8]The plug-in matchmaking of the other TOSCA elements is analogous.

```
1  protected void matchCapabilities() {
2    pluginMatchedCapabilities = new ArrayList<Capability>();
3    List<Capability> sCaps =
4      s.getBoundaryDefinitions().getCapabilities().getList();
5    List<CapabilityDefinition> newUnmatchedCapabilities =
6      new ArrayList<CapabilityDefinition>();
7    boolean matched;
8    for(CapabilityDefinition cDef : unmatchedCapabilities) {
9      matched = false;
10     for(Capability c : sCaps) {
11       matched = match(cDef, c);
12       if(matched) {
13         pluginMatchedCapabilities.add(c);
14         break;
15       }
16     }
17     if(!matched) newUnmatchedCapabilities.add(cDef);
18   }
19   unmatchedCapabilities = newUnmatchedCapabilities;
20   if(unmatchedCapabilities.isEmpty())
21     areCapabilitiesMatched = true;
22 }
23
24 protected boolean match(CapabilityDefinition cDef, Capability c) {
25   if(!cDef.getName().equals(c.getName()))
26     return false;
27   CapabilityType cType = c.getType();
28   while(cType != null) {
29     if(cDef.getCapabilityType().getName().equals(cType.getName()))
30       return true;
31     cType = cType.derivedFrom();
32   }
33   return false;
34 }
```

FIGURE 3.7: *PlugInMatchmaker.matchCapabilities()* method.

comparison is performed by the *match* method (lines 24-34) which checks whether a capability *c* has the same name as capability definition *cDef* (lines 25-26) and whether *c* either has the same type of or a type derived from that of *cDef* (lines 27-32). If no capability matches the capability definition under consideration, then the latter is added to the (new) set of unmatched capability definitions (line 17). After the loop, the set of *unmatchedCapabilities* is properly updated (line 19). If the latter is empty (i.e., if there are no unmatched capability definitions), then the *areCapabilitiesMatched* variable is set to *true* (lines 20-21). Otherwise, the method terminates (by leaving it set to *false*).

Example 3.3. We now use a (toy) example to illustrate the behaviour of our proof-of-concept implementation. Consider the node type *Server* and the service templates[9] *ApacheServer*, *PaaSServer*, and *PaaSServer2* in Fig. 3.8. Suppose that the capability *WSRuntime* of *Server* and *ApacheServer* is of *WSRuntimeCapabilityType*, while those of *PaaSServer* and *PaaSServer2* are of *WebAppCapabilityType* (which is a sub-type of *WSRuntimeCapabilityType*). Suppose also that the type of all requirements is *SWContainerRequirementType*, the type of all properties is *String*, and all service templates expose a *HighAvailabilityPolicy* which is applicable to *Server*.

[9]For the sake of readability, in Fig. 3.8 we abstract from the internal topology of the service templates.

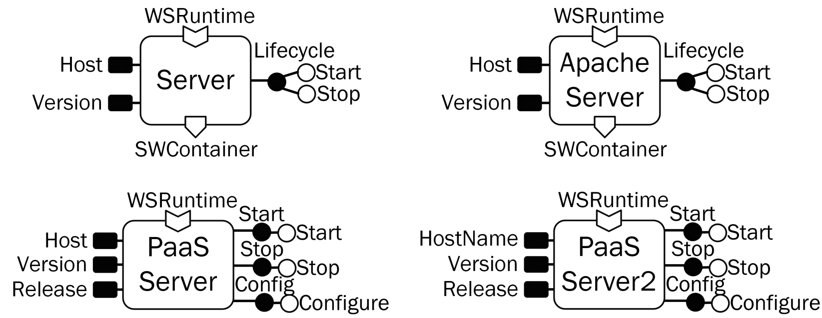


FIGURE 3.8: An example of node type (*Server*), and three examples of service templates (*ApacheServer*, *PaaSServer*, *PaaSServer2*).

Please note that the example is built in such a way that, according to Defs. 3.1 and 3.6, all possible situations are covered:

$$ApacheServer \equiv Server \wedge PaaSServer \not\equiv Server \wedge$$
$$PaaSServer \simeq Server \wedge PaaSServer2 \not\simeq Server.$$

We can easily develop a unit-test class[10] which let us obtain the above mentioned results (Fig. 3.9) by employing the *ExactMatchmaker* and *PlugInMatchmaker* previously introduced. □

```
Console
<terminated> Example
ApacheServer   exactly matches           Server
PaaSServer     does not exactly match    Server
PaaSServer     plug-in matches           Server
PaaSServer2    does not plug-in match    Server
```

FIGURE 3.9: Snapshot of the matchmaking results.

Further remarks

Thanks to the way in which the *match()* methods were implemented (Figs. 3.4 and 3.6), the *PlugInMatchmaker* can be directly exploited to determine whether a service template exactly or plug-in matches a node type. Suppose for instance that *areCapabilitiesMatched* is *true*. If *plugInMatchedCapabilities* is empty, then capabilities were exactly matched. Otherwise, they were plug-in matched. The same holds for requirements, policies, properties, and interface operations.

Additionally, the information in the fields of *PlugInMatchmaker* can also be employed to automate the adaptation of a matched service template. Fig. 3.10 shows the pseudo-code of a method to be included in the *PlugInMatchmaker* class in order to automatically adapt a service template s that plug-in matches a node type n.

[10]The source code of the example is available at https://github.com/jacopogiallo/Finding-available-services-in-TOSCA-compliant-clouds/blob/master/src/di/unipi/example/Example.java.

```
1  ServiceTemplate getAdaptation() {
2    //Creation of the adapted ServiceTemplate
3    ServiceTemplate adapted = new ServiceTemplate();
4    adapted.topology.addNode(s);
5
6    //Adaptation of capabilities
7    matchedCapabilities = exactMatchedCapabilities U plugInMatchedCapabilities;
8    for cap in matchedCapabilities { adapted.boundaries.expose(cap); }
9
10   //Adaptation of requirements
11   if plugInMatchedRequirements.isEmpty() {
12     for req in exactMatchedRequirements { adapted.boundaries.expose(req); }
13   }
14   else {
15     NodeTemplate echo = new NodeTemplate();
16     adapted.topology.addNode(echo);
17     adapted.topology.addRelationshipFromTo(s,echo);
18     for req in n.getRequirements() {
19       echo.addRequirement(req);
20       adapted.boundaries.expose(req);
21     }
22   }
23
24   //Adaptation of policies
25   for pol in exactMatchedPolicies { adapted.boundaries.expose(pol) }
26
27   //Adaptation of properties
28   matchedProperties = exactMatchedProperties U plugInMatchedProperties;
29   for prop in matchedProperties { adapted.boundaries.expose(prop) }
30
31   //Adaptation of interfaces
32   matchedInterfaces = exactMatchedInterfaces U plugInMatchedInterfaces;
33   for intf in matchedInterfaces { adapted.boundaries.expose(intf) }
34
35   return adapted;
36 }
```

FIGURE 3.10: Pseudo-code of the adaptation of *plug-in* matched services.

The proposed implementation can be fruitfully employed by the user also in case of no matching. Since the matchmaking is performed in a "verbose" way (i.e., instead of halting when a condition is not satisfied, it always checks all conditions and properly instantiates the corresponding fields), the collected information can be fruitfully exploited to manually adapt an available service template. In this respect, a methodology to manually adapt plug-in unmatched services (if possible) is described in the following section.

3.2 Overcoming naming differences

Example 3.2 illustrated that a service template S may fail to plug-in match a node type N only because of syntactically different names for compatible features, while a less strict definition of matching should allow S to match also N. We now define two other types of matching (*renaming-based* and *white-box*), each permitting to ignore larger sets of naming differences when type-checking service templates with respect to node types. Finally, we show how to avoid the usage of ontologies by providing a methodology

for adapting plug-in unmatched service templates which is based upon the notions of renaming-based and white-box matching.

3.2.1 Renaming-based matching

We now further extend the definition of matching of a service template with a node type in order to ignore naming differences (i.e., by permitting to match features whose names are syntactically different, but semantically equivalent). Since the semantics of policies depends only on types, we extend plug-in matching (Def. 3.6) only on capabilities, requirements, properties and interfaces.

Definition 3.10 (Renaming-based matching)**.** *A service template S* renaming-based *matches a node type N ($S \sim N$) iff:*

$$
\begin{array}{llll}
(1) & \mathsf{Reqs}(S) & \sim_R & \mathsf{Reqs}(N) \quad \textit{and} \\
(2) & \mathsf{Caps}(S) & \sim_C & \mathsf{Caps}(N) \quad \textit{and} \\
(3) & \mathsf{Pols}(S) & \equiv_{PO} & \mathsf{Pols}(N) \quad \textit{and} \\
(4) & \mathsf{Props}(S) & \sim_{PR} & \mathsf{Props}(N) \quad \textit{and} \\
(5) & \mathsf{Ints}(S) & \sim_I & \mathsf{Ints}(N).
\end{array}
$$

Intuitively speaking, a service template has to expose "less" requirements than a node type. Names of requirements can be semantically equivalent, and types of requirements do not need to strictly coincide.

Notation 3.3. *Let n_1 and n_2 be the names of two TOSCA definitions. We will write $n_1 \bowtie n_2$ to denote that names n_1 and n_2 are semantically equivalent*[11].

Definition 3.11 (Renaming-based matching of requirements)**.** *Let N be a node type and let S be a service template. Then:*

$$
\begin{array}{l}
\mathsf{Reqs}(S) \sim_R \mathsf{Reqs}(N) \textit{ iff} \\
\quad \forall r_S \in \mathsf{Reqs}(S)\ \exists r_N \in \mathsf{Reqs}(N) : \mathsf{name}(r_N) \bowtie \mathsf{name}(r_S) \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{type}(r_N) \geq \mathsf{type}(r_S).
\end{array}
$$

A service template must expose all capabilities of a node type. Names of capabilities can be semantically equivalent, and types of capabilities do not need to stricly coincide. The same holds for properties.

Definition 3.12 (Renaming-based matching of capabilities and properties)**.** *Let N be a node type and S a service template. Then:*

[11]The semantical equivalence of syntactically different names may be implemented by employing ontology-based descriptions of cloud service functionalities (e.g., as proposed by O'Sullivan and Lewis [99]). Namely, TOSCA node types and service templates may include ontology-based annotations associated with the names of their capabilities, requirements, properties and operations. Instead of assuming that all TOSCA-based cloud application specifications are ontology-annotated, we will describe an ontology-free methodology for adapting a service template S that renaming-based or white-box matches a desired node type N so as to exactly match N (see Sect. 3.2).

$\mathsf{Caps}(S) \sim_C \mathsf{Caps}(N)$ *iff*

$$\forall c_N \in \mathsf{Caps}(N)\ \exists c_S \in \mathsf{Caps}(S) : \mathsf{name}(c_S) \bowtie \mathsf{name}(c_N) \wedge \mathsf{type}(c_S) \geq \mathsf{type}(c_N).$$

$\mathsf{Props}(S) \sim_{PR} \mathsf{Props}(N)$ *iff*

$$\forall p_N \in \mathsf{Props}(N)\ \exists p_S \in \mathsf{Props}(S) : \mathsf{name}(p_S) \bowtie \mathsf{name}(p_N) \wedge \mathsf{type}(p_S) \geq \mathsf{type}(p_N).$$

A service template must also expose all the operations exposed by a node type. Names of operations can be ignored, while names of operation parameters can be semantically equivalent and their types do not need to strictly coincide.

Definition 3.13 (Renaming-based matching of interfaces)**.** *Let N be a node type and let S be a service template. Then:*

$\mathsf{Ints}(S) \simeq_I \mathsf{Ints}(N)$ *iff*

$$\forall i_N \in \mathsf{Ints}(N), o_N \in \mathsf{Ops}(i_N) : \exists i_S \in \mathsf{Ints}(S), o_S \in \mathsf{Ops}(i_S) :\ o_S \sim_o o_N.$$

where $o_x \sim_o o_y$ *iff*

$|\mathsf{I}(o_x)| = |\mathsf{I}(o_y)|$ *and*

$|\mathsf{O}(o_x)| = |\mathsf{O}(o_y)|$ *and*

$\forall a \in \mathsf{I}(o_x), \exists! b \in \mathsf{I}(o_y) : \mathsf{name}(a) \bowtie \mathsf{name}(b) \wedge \mathsf{type}(b) \geq \mathsf{type}(a)$ *and*

$\forall a \in \mathsf{O}(o_x), \exists! b \in \mathsf{O}(o_y) : \mathsf{name}(a) \bowtie \mathsf{name}(b) \wedge \mathsf{type}(a) \geq \mathsf{type}(b).$

where $\mathsf{I}(o)$ *and* $\mathsf{O}(o)$ *denote the input and output parameters of operation o.*

In Sect. 3.1.2 we have illustrated how a service template S that plug-in matches a node type can be easily adapted so as to exactly match that node type. The same holds for renaming-based matching: A service template S that renaming-based matches a node type can be easily adapted into a new service template S' that exactly matches that node type. As for the case of plug-in matching, S' is built by creating a new service template having S as its only node, and by simply exposing (via the boundary definitions) the capabilities, policies, properties, and interfaces of the node type to be matched. If requirements renaming-based match (but do not exactly match) then a dummy *NoBe* node is introduced to artificially extend the set of requirements of S so as to expose the same requirements of the node type to be matched. Moreover, differently from plug-in adaptation, renaming-based adaptation may rename properties, interfaces, operations, and operation parameters.

Example 3.4. Example 3.2 illustrated a service template S that does not plug-in match a node type N_3 since S exposes a property $p1$ different from the property pA exposed by N_3. It is easy to see that Def. 3.12 permits S to renaming-based match N_3 (viz., $S \sim N_3$) if the type of $p1$ extends or is equal to the type of pA and if $p1$ and pA —even if syntactically different— refer to the same property (viz., $\mathsf{name}(p1) \bowtie \mathsf{name}(pA)$). Fig. 3.11 illustrates how S can be adapted so as to exactly

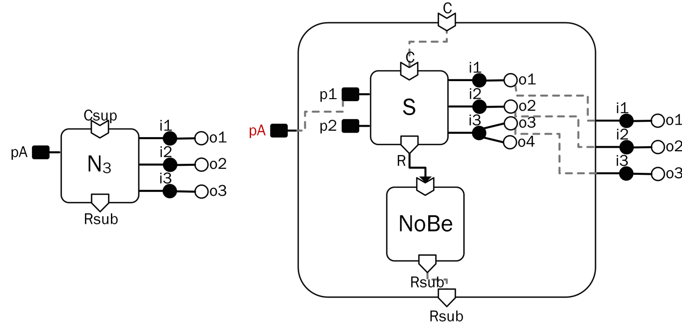


FIGURE 3.11: Renaming-based matching example.

match N_3, by letting the new service template S' expose also the renamed property pA. □

Example 3.5. Suppose that a cloud application developer needs to employ a node type OS (Fig. 3.12), whose interface M exposes the following operations

$$Start : \{\} \rightarrow \{\},\ InstallPkg : \{name\} \rightarrow \{succeded\}, \text{ and } Shutdown : \{\} \rightarrow \{\}.$$

Suppose also that a service template $UbuntuOS$ is available, and that it exhibits a management interface U featuring the following operations:

$$Start : \{\} \rightarrow \{\},\ Shutdown : \{\} \rightarrow \{\},\ Retrieve : \{pkgName\} \rightarrow \{url\},$$
$$Download : \{url\} \rightarrow \{sourcePath\}, \text{ and } Install : \{sourcePath\} \rightarrow \{installed\},$$

with $name \bowtie pkgName$ and $succeded \bowtie installed$. For the sake of simplicity we also assume that $\mathsf{name}(x) \bowtie \mathsf{name}(y)$ implies $\mathsf{type}(x) = \mathsf{type}(y)$.

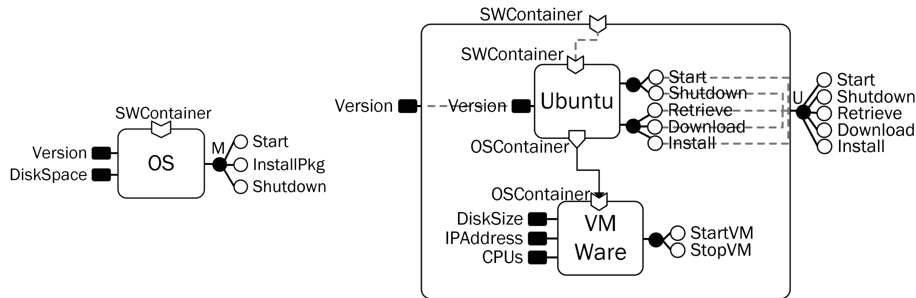


FIGURE 3.12: A service template that does not renaming-based match a node type.

It is easy to see that while $UbuntuOS$ capabilities exactly match OS capabilities, $UbuntuOS$ properties and interfaces do not renaming-based match OS's ones. This is because OS is exposing a property ($DiskSpace$) that $UbuntuOS$ is not featuring, and because $UbuntuOS$ does not offer the operation $InstallPkg$ exposed by OS. Still, one may observe that property $DiskSpace$ may correspond to one of the properties of an internal node of $UbuntuOS$ (i.e., to $VM\ Ware$'s property $Disk Size$) and that operation $InstallPkg$ might be offered by $UbuntuOS$ by suitably combining some of its operations. This suggests that a "white-box" definition of matching could allow the service template $UbuntuOS$ to match the desired node type OS (as we will discuss in the next section). □

3.2.2 White-box matching

A service template S that does not renaming-based match a node type because of some missing requirement, capability, property, or operation, may actually include such missing elements internally, without exposing them on its boundaries.

As for the previous definitions of matching, the following definition specifies when a service template S *white-box* matches a node type N in terms of the requirements, capabilities, policies, properties and interfaces of S and N. As we already observed in Sect. 3.2.1, intuitively speaking, a node type N must expose (at least) a set of requirements which are semantically equivalent to (all) those of the service template S. Moreover, node types do not specify concrete policies. For these reasons, the following definition extends Def. 3.10 only on capabilities, properties and interfaces.

Definition 3.14 (White-box matching). *A service template S* white-box *matches a node type N ($S \square N$) iff:*

$$\begin{array}{llcll} (1) & \mathsf{Reqs}(S) & \sim_R & \mathsf{Reqs}(N) & \mathit{and} \\ (2) & \mathsf{Caps}(S) & \square_C & \mathsf{Caps}(N) & \mathit{and} \\ (3) & \mathsf{Pols}(S) & \equiv_{PO} & \mathsf{Pols}(N) & \mathit{and} \\ (4) & \mathsf{Props}(S) & \square_{PR} & \mathsf{Props}(N) & \mathit{and} \\ (5) & \mathsf{Ints}(S) & \square_I & \mathsf{Ints}(N). & \end{array}$$

The following definition extends the matching of capabilities and properties (Def. 3.12) to consider also the internals of a service template (viz., the node templates and relationship templates in its topology).

Notation 3.4. *Let S be a service template, and let E be a node template or a relationship template. We denote by $S \rightarrow E$ the fact that E is an element of the (internal) topology of S.*

Definition 3.15 (White-box matching of capabilities and properties). *Let N be a node type and let S be a service template. Then:*

$$\begin{aligned} &\mathsf{Caps}(S) \square_C \mathsf{Caps}(N) \textit{ iff} \\ &\quad \forall c_N \in \mathsf{Caps}(N)\ \exists c_S : (c_S \in \mathsf{Caps}(S) \vee (\exists E : S \rightarrow E \wedge c_S \in \mathsf{Caps}(E))) \wedge \\ &\qquad\qquad (\mathsf{name}(c_S) \bowtie \mathsf{name}(c_N) \wedge \mathsf{type}(c_S) \geq \mathsf{type}(c_N)). \end{aligned}$$

$$\begin{aligned} &\mathsf{Props}(S) \square_{PR} \mathsf{Props}(N) \textit{ iff} \\ &\quad \forall p_N \in \mathsf{Props}(N)\ \exists p_S : (p_S \in \mathsf{Props}(S) \vee (\exists E : S \rightarrow E \wedge p_S \in \mathsf{Props}(E))) \wedge \\ &\qquad\qquad (\mathsf{name}(p_S) \bowtie \mathsf{name}(p_N) \wedge \mathsf{type}(p_S) \geq \mathsf{type}(p_N)). \end{aligned}$$

The following definition extends the matching of operations (Def. 3.13) to consider also operations that a service template can feature by combining its operations in a suitable plan.

Definition 3.16 (White-box matching of interfaces)**.** *Let N be a node type, let S be a service template, and let $\Pi(S)$ the set of all possible plans combining the operations of S. Then:*

$$\begin{array}{l}\mathsf{Ints}(S) \sqsupset_I \mathsf{Ints}(N) \textit{ iff} \\ \quad \forall i_N \in \mathsf{Ints}(N), o_N \in \mathsf{Ops}(i_N) : \ (\exists i_S \in \mathsf{Ints}(S), o_S \in \mathsf{Ops}(i_S) : \ o_S \sim_o o_N) \vee \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad (\exists p : p \in \Pi(S) \wedge [p] \sim_o o_N)\end{array}$$

where $[p]$ denotes an operation whose input and output parameters model the overall input-output behaviour of the plan p.

The existence of a plan that suitably combines a set of available operations to obtain an input-output behaviour equivalent to a desired operation can be determined by adapting the (ontology-aware) discovery function proposed by Brogi et al. [30].

```
1:  function FINDOPERATIONS(available, t, selected)
2:      requiredPars ← O(t) ∪ {p | p ∈ I(o) ∧ o ∈ selected}
3:      availablePars ← O(t) ∪ {p | p ∈ O(o) ∧ o ∈ selected}
4:      missingPars ← {p | p ∈ inputs ∧ ∄p′ ∈ outputs : p′ ▷ p}
5:      if missingPars = ∅ then return { selected }
6:      else
7:          p ← choose(missingPars)
8:          candidates ← {o ∈ available | ∃p′ ∈ O(o) : p′ ▷ p}
9:          res ← ∅
10:         for all o ∈ candidates do
11:             selected′ ← selected ∪ {o}
12:             if minimal(selected, op) then
13:                 res ← res ∪ FINDOPERATIONS(available, t, selected′)
14:         return res
```

(where $p' \triangleright p$ stands for $\mathsf{name}(p') \bowtie \mathsf{name}(p)$ and $\mathsf{type}(p') \geq \mathsf{type}(p)$)

FIGURE 3.13: Function to discover sets of available operations that can be composed into plans featuring the input-output behaviour of a target operation.

The FINDOPERATIONS function (Fig. 3.13), given a set of operations, returns all subsets of such operations that can be composed into a plan featuring the input-output behaviour of a target operation. The function inputs a set of $available$ operations, the target operation t to be simulated, and a (initially empty) set of $selected$ operations. First, the function computes the set $requiredPars$ of required parameters (viz., the output parameters of the target operation t, as well as the parameters required as inputs by the currently $selected$ operations — line 2), and the set $availablePars$ of available parameters (viz., the input parameters of the target operation t, as well as the outputs of the currently $selected$ operations — line 3). It then computes the set $missingPars$ of parameters that are required but not yet available as follows: A required parameter p is included in $missingPars$ only if there is no parameter p' in $availablePars$ such that p' is "equal to or more general

than" p (line 4). If there are no missing parameters to be generated the current set of $selected$ operations is returned (line 5). Otherwise, a missing output p is non-deterministically chosen (line 7). The function then computes the set $candidates$ of available operations that produce an output equal to or more general than p (line 8), and creates an initially empty variable res where to accumulate the results (line 9). For each operation o in $candidates$, a new set $selected'$ of operations is computed (by adding o to the current set of $selected$ operations — line 11). If $selected'$ is minimal[12], then the function recurs on it, and updates the set res of computed results by adding the result of the recursion (line 13). Finally, the function returns the computed results (line 14)[13].

Finally, it is worth highlighting that when a service template S white-box matches a node type N then S can be adapted into a new service template S' that exactly matches that node type. Differently from the cases of plug-in and renaming-based matching, the boundary definitions of S are first extended in order to expose the capabilities, properties or plans internal to S that were detected by the white-box matching. The obtained service template S_{tmp} renaming-based matches node type N, and the adaptation described in Sect. 3.2.1 can be now applied to build a service template S' having S_{tmp} as its only node, and by simply exposing (via the boundary definitions) the capabilities, policies, properties, and interfaces of the node type N to be matched. If requirements plug-in match (but do not exactly match) then a dummy node is introduced to artificially extend the set of requirements of S so as to expose the same requirements of the node type to be matched.

Example 3.6. Example 3.5 illustrated a service template (viz., $UbuntuOS$) that cannot renaming-based match a node type (viz., OS) since the latter exposes one property more than the former (i.e., $DiskSpace$), and since $UbuntuOS$ does not offer the operation $InstallPkg$. Def. 3.14 permits $UbuntuOS$ to white-box match OS (viz., $UbuntuOS \square OS$) if, for instance, property $DiskSize$ of node $VMWare$ of $UbuntuOS$ is semantically equivalent to property $DiskSpace$ of OS, and if there exists a plan P combining some $UbuntuOS$'s operations, whose input-output behaviour simulates that of the operation $InstallPkg$ (viz., $[P] \sim_o InstallPkg$). It is easy to observe that function FINDOPERATIONS returns a minimal set of operations of $UbuntuOS$ that can simulate $InstallPkg$, namely $\{Retrieve, Download, Install\}$. Such set can then be used to build a plan P simulating the input-output behaviour of the desired operation $InstallPkg$:

$$P = Retrieve \cdot Download \cdot Install$$

[12]We do not include here the definition of the $minimal$ function, which can be found in Brogi et al. [30]. Intuitively speaking, a set O of operations can simulate the input-output behaviour of an operation o iff
(1) $\forall x \in \mathsf{O}(o)\ \exists y \in \bigcup_{o' \in O} \mathsf{O}(o') : y \triangleright x$, and
(2) $\forall y \in \bigcup_{o' \in O} \mathsf{I}(o')\ \exists x \in (\bigcup_{o' \in O} \mathsf{O}(o') \cup \mathsf{I}(o)) : x \triangleright y$.
The set O is also minimal with respect to o iff $\nexists O' \subset O$ that can simulate o.

[13]The function can be proved to be terminating, sound, and complete [108].

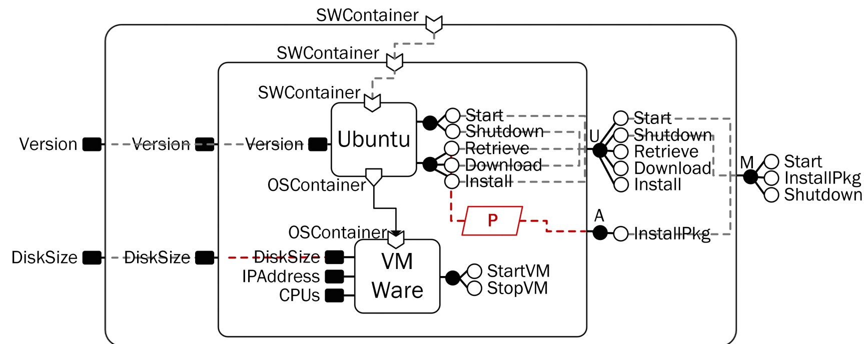


FIGURE 3.14: White-box adaptation of a *ServiceTemplate*.

Fig. 3.12 illustrates the adaptation of *UbuntuOS*. Its boundary definitions are first extended to expose property *DiskSize* of node *VM Ware* as property *DiskSpace*, and to expose the plan P as operation *InstallPkg*. Then, the resulting service template is encapsulated into a new service template so as to expose only the capabilities, properties, and interfaces of the node type *OS* to be matched. □

3.2.3 Adaptation of matched cloud applications

The *renaming-based* and *white-box* matching defined in Sects. 3.2.1 and 3.2.2 may be implemented by employing ontology-based descriptions of cloud services [99]. To avoid ontology-related problems (e.g., cross-ontology matchmaking [75, 85]), in this section we propose a methodology to manually adapt plug-in unmatched service templates so as to exactly match a target node type. Namely, we show how to exactly match target node types by non-intrusively adapting renaming-based matched service templates, and by intrusively adapting white-box matched service templates.

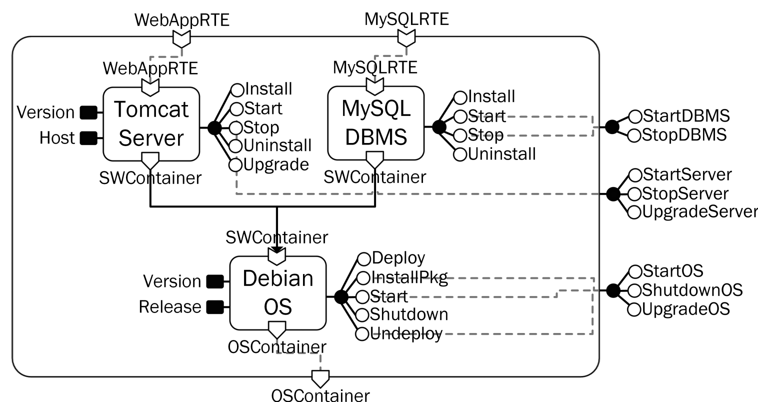


FIGURE 3.15: Available service template *WebAppEnvironment*.

In doing so, we also provide some examples showing how to adapt the service template *WebAppEnviroment* (Fig. 3.15) to exactly match different node types. For the sake of simplicity, we will assume that each capability or requirement is of an homonym capability type or requirement type

(e.g., the capability *WebAppRTE* is of *WebAppRTE* capability type, the requirement *SWContainer* is of *SWContainer* requirement type, etc.). We will also assume that all properties are strings, and that all operations have no input parameters and return a boolean parameter witnessing whether they successfully completed (e.g., the *TomcatServer*'s operation *Start*, as well as the operation *StartServer* on the boundaries, return a parameter *TomcatServerStarted*, which is true if the *TomcatServer* has correctly started, and false otherwise). Finally, we will abstract from policies, as they just require to check whether they are applicable to a node type.

Adapting renaming-based matched service templates

As we illustrated in Sect. 3.2.1, a service template S renaming-based matches a target node type N when the plug-in matching fails only because of naming differences. If S renaming-based matches N, then the former can be adapted into a new service template which exactly matches N. The adaptation consists in building a new service template which contains the available one as a node template and whose boundaries are built by declaring the same features of N and by mapping each of them to the matched feature of S. This can be done automatically if we employ ontologies, otherwise we need the manual intervention of the application designer.

We now illustrate how the application designer may non-intrusively adapt a service template S which does not plug-in match a node type N into a new service template S' which exactly matches N. Fig. 3.16 describes how such an adaptation can be successfully performed when S exposes all capabilities, properties, interface operations and requirements as N, but in a syntactically different way.

The adaptation described in Fig. 3.16 implements the relaxed matching conditions of Def. 3.10 (in terms of ontology-based name equivalences). It is worth noting that, according to the definition of renaming-based matching, the adaptation process cannot succeed if capability, property, operation or requirement mismatches are not just syntactic (i.e., if they are not only due to names which are syntactically different but semantically equivalent). More precisely, the adaptation described in Fig. 3.16 will fail if one of the steps cannot be performed, while it succeed if all the steps are performed.

Example 3.7. Consider the target node type *WebEnv* in Fig. 3.17, where the capabilities *WebAppRuntime* and *MySQLRuntime* are respectively of type *WebAppRTE* and *MySQLRTE*, and where both requirements are of type *OSContainer*. All operations are without input parameters, and return a boolean parameter witnessing whether they successfully completed (e.g., the operation *StartWebAppRuntime* returns a parameter *WebAppRuntimeStarted*, which is true if the *WebAppRuntime* capability is concretely provided, and false otherwise). We observe that, according to Def. 3.10, the available service template *WebAppEnvironment* (Fig. 3.15) renaming-based matches the target node type *WebEnv*.

(1) Create the adapted service template S' which initially contains S as the only node template in its topology.

(2) For each capability (property) exposed by N

   (a) define a capability (property) with the same name and type on the boundaries of S', and

   (b) map the defined capability (property) onto the corresponding one of S.

(3) For each interface exposed by N, define an interface with the same name on the boundaries of S'. Then, for each operation o exposed by (an interface of) N

   (a) define an operation with the same name and parameters in the corresponding interface exposed by S', and

   (b) map o onto an operation of S which is semantically equivalent to o.

(4) Add a dummy node template *NoBe* (whose capabilities satisfy the requirements of S and whose requirements are the same of N) to the topology of S'. Then, for each requirement exposed by N

   (a) define a requirement with the same name and type on the boundaries of S', and

   (b) map the defined requirement to the corresponding one of *NoBe*.

(where mapping f onto f' simply means that f is a reference to f')

FIGURE 3.16: A methodology for adapting renaming-based matched service templates.

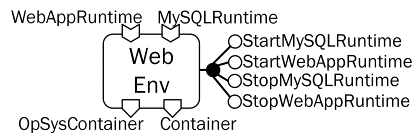


FIGURE 3.17: Target node type *WebEnv*.

Figs. 3.18, 3.19 and 3.20 illustrate how *WebAppEnvironment* can be adapted so as to exactly match *WebEnv*. First, (1) we create a new service template which contains *WebAppEnvironment* as the only node template (Fig. 3.18).

Since *WebAppRTE* and *WebAppRuntime*, as well as *MySQLRTE* and *MySQL-Runtime*, are of the same capability type, (2) we adapt the capabilities by adding *WebAppRuntime* and *MySQLRuntime* to the boundaries of the adapted service template and by mapping them to the corresponding capabilities of the available service template (i.e., *WebAppEnvironment* and *MySQLRuntime*). Analogously, (3) we adapt the operations *StartMySQLRuntime*, *StartWebAppRuntime*, *StopMySQLRuntime*, and *StopWebAppRuntime*, by mapping them to the corresponding operations of *WebAppEnvironment* (i.e., *StartDBMS*, *StartServer*, *StopDBMS*, and *StopServer* — Fig. 3.19).

Finally, (4) we artificially extend the requirements of the available service template (i.e., *WebAppEnvironment*) to exactly match those of target node type (i.e., *WebEnv*). Namely, we add a dummy node template *NoBe* (whose capabilities satisfy the requirements of *WebAppEnvironment* and whose requirements are the same of *WebEnv*) to the topology of the adapted service template, we define the same requirements of the target node type on the boundaries of the adapted service template, and we map each of them to the corresponding one of *NoBe* (Fig. 3.20).

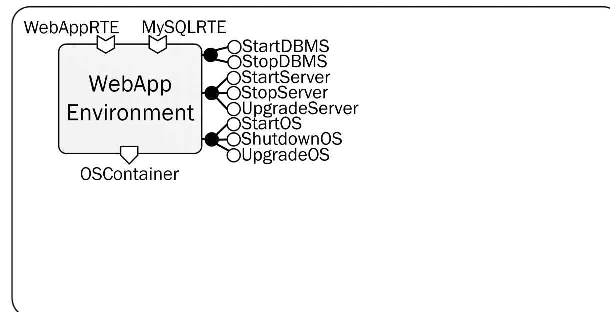


FIGURE 3.18: An example on how to apply the step 1 of the (non-intrusive) adaptation methodology.

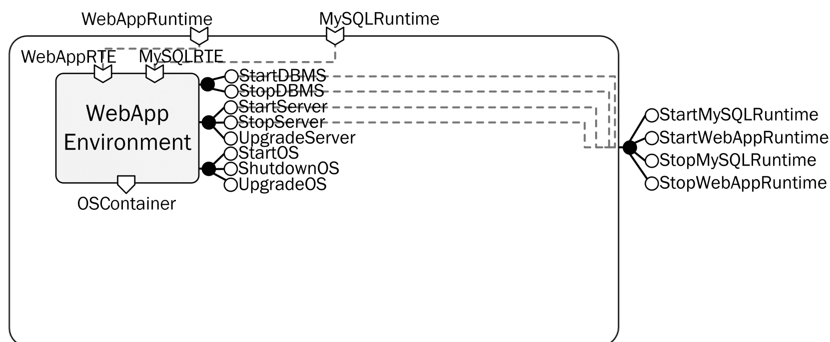


FIGURE 3.19: An example on how to apply the steps 2 and 3 of the (non-intrusive) adaptation methodology.

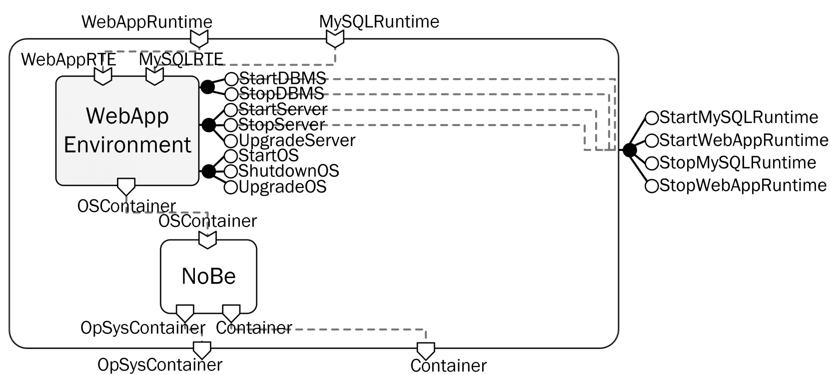


FIGURE 3.20: An example on how to apply step 4 of the (non-intrusive) adaptation methodology.

The obtained service template (Fig. 3.20) exposes all the features exhibited by the target node type *WebEnv*. This implies that they exactly match, and subsequently that the adapted service template can be employed to instantiate the node type *WebEnv*. □

Adapting white-box matched service templates

White-box matching relaxes renaming-based matching by extending the feature research to the internal elements of a service template, and by allowing to combine internal operations in order to obtain plans which are semantically equivalent to missing operations (see Sect. 3.2.2). Furthermore, if S white-box matches N, then the former can be adapted into a new service template which exactly matches N. The adaptation consists of building a new service template which contains the available one as a node template, and whose boundaries are built by declaring the same features of N and by mapping each one of them to the matched feature of S. This can be done automatically if we employ ontologies, otherwise the application designer needs to manually adapt S.

We hereby illustrate how an application designer may intrusively adapt a service template S which does not renaming-based match a node type N into a new service template S' which exactly matches N. Fig. 3.21 describes how such an adaptation can be successfully performed when (i) S exposes all capabilities, properties, and requirements as N, but internally and/or in a syntactically different way, and (ii) when N features one or more interface operation which is not matched by any operation featured by S, while it can be matched by some composition of the internal operations of S.

According to the definition of white-box matching, the adaptation process cannot succeed if missing capabilities and properties cannot be matched internally as well as if operation mismatches cannot be solved by composing internal operations. Namely, the adaptation described in Fig. 3.21 will fail if one of the steps cannot be performed, while it succeed if all the steps are performed.

Example 3.8. Consider the target node type *IntegratedWebEnv* in Fig. 3.22, where the capabilities *WebAppRuntime* and *MySQLRuntime* are respectively of type *WebAppRTE* and *MySQLRTE*, and where both requirements are of type *OSContainer*. Both operations are without input parameters, and return two boolean parameters witnessing whether they successfully completed (e.g., the operation *Start* returns two parameters *WebAppRuntimeStarted* and *MySQLRuntimeStarted*, each of which is true if the corresponding capability is concretely provided, and false otherwise). We observe that, according to Def. 3.14, the available service template *WebAppEnvironment* (Fig. 3.15) white-box matches the target node type *IntegratedWebEnv*.

Fig. 3.23 illustrates the service template obtained by applying the adaptation process discussed above to the service template *WebAppEnvironment* (to exactly match the target node type *IntegratedWebEnvironment*). Namely, we first create a

(1) Create the adapted service template S' which initially contains S as the only node template in its topology.

(2) For each capability (property) c exposed by N

(a) define a capability (property) with the same name and type on the boundaries of S',

(b) if c does not correspond to any capability (property) exposed by S, search inside of the topology of S a capability (property) corresponding to c and expose it on the boundaries of S, and

(c) map such capability (property) to the corresponding one exposed by S.

(3) For each interface exposed by N, define an interface with the same name on the boundaries of S'. Then, for each operation o exposed by N,

(a) define an operation with the same name and parameters in the corresponding interface exposed by S', and

(b) map the defined operation to

- an operation of S which is semantically equivalent to o, or
- a (new) operation o_{ex} of S which is suitably extracted from its internal definitions. With "suitably extracted" we mean that (i) a new interface has been defined on the boundaries of S, and (ii) o_{ex} has been added to its operations, and (iii) o_{ex} has been mapped either to an internal operation of S which is considered semantically equivalent to o, or to a plan which combines the internal operations of S to obtain an operation which is semantically equivalent to o.

(4) Add a dummy node template *NoBe* (whose capabilities satisfy the requirements of S and whose requirements are the same of N) to the topology of S'. Then, for each requirement exposed by N

(a) define a requirement with the same name and type on the boundaries of S', and

(b) map the defined requirement to the corresponding one of *NoBe*.

(where mapping f onto f' simply means that f is a reference to f')

FIGURE 3.21: A methodology for adapting white-box matched service templates.

new service template which contains *WebAppEnvironment* as the only node template in its topology. Capabilities and requirements are adapted as shown in Example 3.7. The *HostName* property is extracted from *WebAppEnvironment*'s internal topology and then mapped on the boundaries of the adapted service template. The operation *Start* requires to generate a plan P_{Start} combining the *Start* operations of *TomcatServer* and *MySQLDBMS*, to expose such plan as an operation of *WebAppEnvironment*, and then to expose such operation also on the boundaries of the adapted service template. The adaptation to obtain *Stop* is analogous.

The obtained service template exposes all the features exhibited by the target node type *IntegrateWebEnvironment*. This implies that they exactly match and subsequently that the adapted service template can be employed to instantiate the node type *IntegratedWebEnvironment*. □

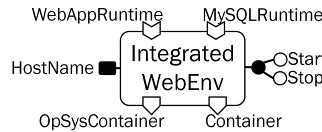


FIGURE 3.22: Target node type *IntegratedWebEnv*.

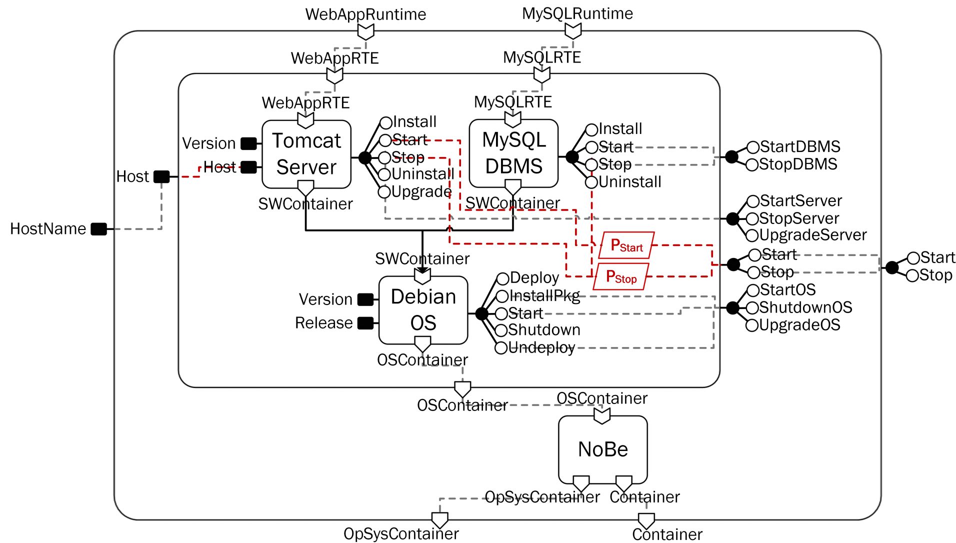


FIGURE 3.23: Example of (intrusive) adaptation of a white-box matched service template.

Remarks

It is important to observe that the adaptation of a TOSCA service template S to (exactly) match a node type N does suffice to reuse any actual service modelled by S to deploy cloud applications that rely on N. This is thanks to the powerful way in which TOSCA supports the deployment of cloud applications. TOSCA permits to pack in a CSAR (*Cloud Service ARchive*) file an application specification together with the actual executable files to be deployed on a cloud platform. When a CSAR file is given in input to a TOSCA container, the latter takes care of deploying and executing the application specification contained in the CSAR file [33, 95]. Therefore, in order to adapt an actual service modelled by a service template S to deploy an application that relies on a node type N, it suffices to adapt S into a new service template S' that matches N — without having to generate an implementation of the adaptation specified by S.

Note that the adaptation works also in the case in which the CSAR of S should not be available, for instance when S is a proprietary service offered by a third party. In such cases it suffices to develop a simple proxy of the remote service modelled by S, and to pack it in a new CSAR file together with the application specification containing S' (and the executable files associated with such specification).

Finally, it is also worth highlighting that, thanks to the features of TOSCA, the simple adaptation methodology described in this chapter considerably reduces the work needed to reuse cloud services if compared with the alternative of explicitly devising adapters as in traditional software adaptation approaches (e.g., the adaptation approaches by Bracciali et al. [23], Guillén et al. [66], Kongdenfha et al. [76]).

Chapter 4

Reusing *fragments* of application topologies

All notions of matching presented in the previous chapter permit reusing cloud applications only in their entirety. This would potentially result in requiring to waste resources to deploy unnecessary software. In this chapter, we show how to overcome this issue by introducing and assessing TOSCA-MART (*TOSCA-based Method for Adapting and Reusing application Topologies*), a method that permits reusing only the fragment of an application topology that is necessary for implementing a desired node.

More precisely, after discussing a running example (also motivating the need for reusing fragments of cloud applications — in Sect. 4.1), in Sect. 4.2 we present the TOSCAMART approach. We then discuss termination, soundness, and time complexity of TOSCAMART in Sect. 4.3. Finally, in Sect. 4.4 we illustrate the feasibility of TOSCAMART by means of a proof-of-concept implementation, and we comparatively assess TOSCAMART with respect to the matching presented in the previous chapter.

4.1 A running example

Suppose that a Web application developer needs to host a PHP application on a cloud environment, along with a MySQL database containing application data. So far, she is required to select the appropriate cloud provider and to explicitly describe the provisioning of her PHP application on this provider. Furthermore, in case she decides to move her application to another provider, this may require to re-describe (and re-implement) the deployment and management of her solution (even from scratch). It would be much better to abstractly describe the desired hosting environment and to provide such description as input to a tool which automatically derives a topology implementing the environment needed to deploy and manage the PHP application.

In TOSCA, this can be done as shown in Fig. 4.1. The environment required to host the PHP and MySQL modules is represented by a node

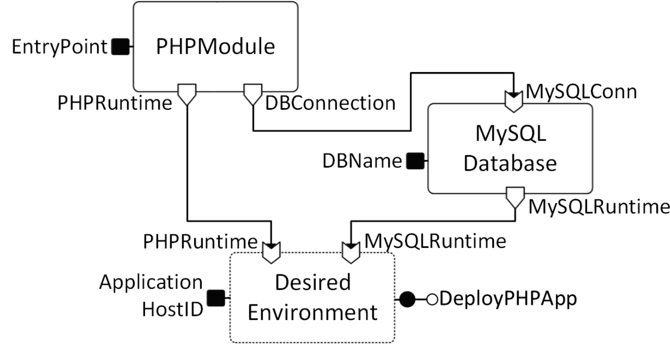


FIGURE 4.1: Motivating scenario.

whose type is *DesiredEnvironment*. This node is used by the application developer to describe the capabilities needed to host her application, to specify that the desired environment must provide an operation for deploying PHP applications, and to instruct that the application host ID must be available as a property. Based on this simple node type, TOSCAMART can derive its implementation by searching among existing cloud applications (as we will see in the next section).

4.2 TOSCAMART

In this section, we illustrate TOSCAMART as a possible solution to derive possible implementations of desired node types by reusing *fragments* of existing cloud application topologies. We first overview the method as a whole, and then illustrate its steps separately.

4.2.1 Overview

Our goal is to derive an implementation for a target node type N by matchmaking and adapting fragments of existing application topologies, taken from a repository $Repo$ of cloud applications. Hence, N[1] and $Repo$ must be input of the TOSCAMART method illustrated in Fig. 4.2.

Once N and $Repo$ are available, each application topology $T_i \in Repo$ is compared with N by employing the MATCHMAKE procedure. As a partial result, we obtain the set $Candidates_{T_i}$, whose elements are

$$\langle T_i, C, \{m_1, m_2, ..., m_n\}\rangle$$

where C is a *candidate fragment* of the topology T_i (i.e., a fragment of T_i whose elements offers all the features declared by N) and m_i is a potential

[1] In the following, we assume that N is defined in such a way that needed features are not redundant (e.g., it is not possible to match more than one capability of N with one of the available capabilities).

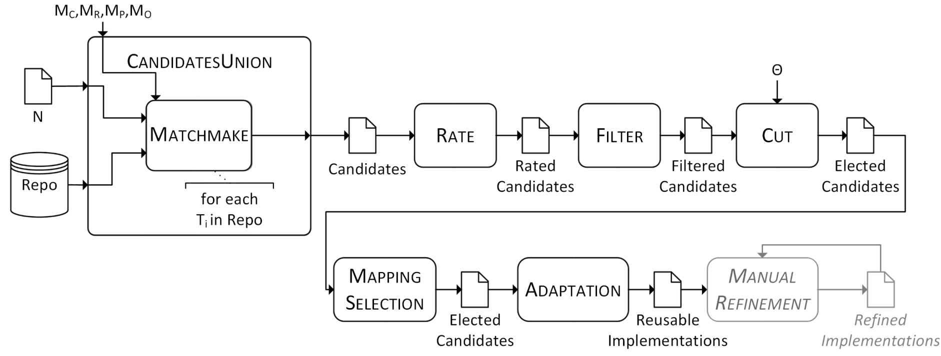


FIGURE 4.2: The TOSCAMART matchmaking and adaptation method.

mapping[2] between the features in N and those in C. Then, all the $Candidates_{T_i}$ are unified to obtain the set $Candidates$, containing all the candidate topology fragments.

Due to the potentially huge number of already available topologies and to the possibility of having multiple candidate fragments for each of these topologies, the set $Candidates$ may become huge. Thus, providing the user with all these candidates is not appropriate. We reduce the number of available candidates by employing three subsequent steps:

(i) RATE computes a $score$ for each candidate using a rating function r. As a result, the set $Candidates$ is transformed in the set $RatedCandidates$, whose elements are $\langle T_i, C, score, \{m_1, m_2, ..., m_n\}\rangle$.

(ii) FILTER reduces the number of $RatedCandidates$ by removing duplicates[3] (i.e., candidates that have the same topology fragment C, the same $score$ and the same sets of potential mappings, independently from the topology T_i they come from).

(iii) CUT reduces the number of candidates according to a threshold Θ. More precisely, the set $FilteredCandidates$ is reduced to the set $ElectedCandidates$, which contains only the "best" Θ candidates (i.e., those having the highest $score$, according to the rating function r).

Each of the $ElectedCandidates$ has to be adapted to properly implement the target N. First, in order to avoid the user to select mappings on her own, we need to select the most proper mapping among the available ones. This is the purpose of the MAPPINGSELECTION step, which can be implemented in various ways (e.g., ontologies, heuristics, compliance rules, etc.).

[2]Notice that a feature of N may be matched by more than one feature of C, e.g., a capability of the desired node N may be matched by different capabilities of different nodes in the topology fragment C. This is why we may have more than one mapping for each candidate topology fragment C.

[3]Duplicates are maintained because they do not significantly impact on the complexity of our approach and they allow the definition of smarter rating functions (e.g., by enabling to count how many times a topology fragment is recurring).

Once the mappings are selected, each of the $ElectedCandidates$ is adapted by resolving the unsatisfied dependencies of the selected components, and by enclosing the candidate fragments into standalone application specifications which implement the target node type N. All these specifications compose the set $ReusableImplementations$, which is the output of the TOSCAMART method. Finally, an optional MANUALREFINEMENT step may be performed to allow developers to manually modify the output node type implementations, if they are not designed as desired.

4.2.2 Repository of application topologies

The repository $Repo$ of application topologies is the knowledge base from which the TOSCAMART method extracts the implementations for the target node type N. Thus, a prerequisite for the applicability of this method is to have a large set of diverse topologies to be included in $Repo$. We include *application models*, which describe components, structure and configuration of applications. TOSCA application models can be retrieved from modelling environments (e.g, Winery [78]), as well as from configuration management systems. For instance, Wettinger et al. [114] show how descriptions used by configuration management systems, such as Chef, can be wrapped into TOSCA application models. We can also include applications already operated in organizations (i.e., *application instances*). Despite such instances are usually not available as topology models, we can cope with them by employing Enterprise Topology Graphs (ETG) [20]. An ETG is a technically-detailed instance model that represents a snapshot of one or multiple applications, including all components, configuration and their relations. Binz et al. [16] shows how to semi-automatically create complete and technically detailed ETGs from existing enterprise applications. These ETGs can then be transformed into TOSCA topologies (as shown by Binz et al. [17]) so as to include them into $Repo$.

For instance, the repository $Repo$ can be populated with the concrete application topologies illustrated in Fig. 4.3, namely with three instantiable topologies implementing (a) a *Moodle* application, (b) a *Wiki* application, and (c) a *SendSMS* web service. In the following, we will show how the TOSCAMART method exploits this repository of instantiable topologies to derive an implementation for the *DesiredEnvironment* in Fig. 4.1. It is easy to see that the former two topologies are offering the desired features (i.e., the *Moodle* application offers all the features via the *ApachePHPModule*, *ApacheWebServer* and *MySQLDBMS* components, while the *Wiki* application offers them in a more integrated fashion via the *XAMPP* server). Thus, TOSCAMART has to detect that both can be reused to implement the desired node, and (by supposing $\Theta = 1$) it also has to return only the adaptation of that having the highest rating. On the other hand, the topology implementing

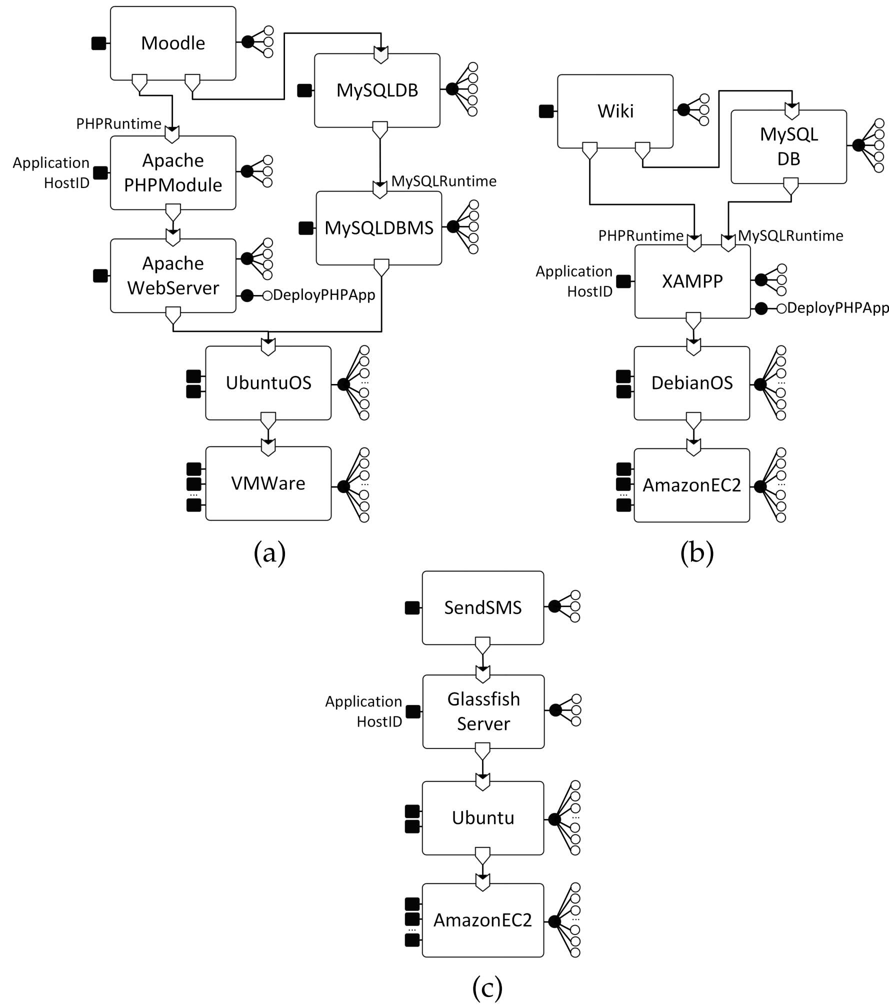


FIGURE 4.3: Three examples of application topologies that can be included in *Repo*.

the *SendSMS* web service is offering only one of the desired features (i.e., the *ApplicationHostID* property via the *GlassfishServer*). Thus, TOSCAMART has to discard it during the matchmaking phase (since it cannot be reused to implement the *DesiredEnvironment*).

4.2.3 Finding the candidate topology fragments

To determine the *fragments* that can be reused to implement a desired node type, we apply the function MATCHMAKE to each topology $T_i \in Repo$, as shown in Fig. 4.2. This allows us to detect the sets $Candidates_{T_i}$, which are then unified by CANDIDATESUNION to obtain $Candidates$, the set of all candidate topology fragments.

The pseudocode of the MATCHMAKE function is listed in Fig. 4.4. Given a node type N and a topology T, it employs the function MATCHCAPS to check whether all the capabilities declared by N (viz., Caps(N)) can be matched by those offered by the components of T (viz., Caps(T)), according to the matchmaking operator M_C (e.g., M_C may be $\equiv_C$ or $\simeq_C$ — see

```
1:  function MATCHMAKE(N, T, M_C, M_R, M_P, M_O)
2:      mCaps = MATCHCAPS(M_C, {}, Caps(N), Caps(T), {})
3:      if Caps(N) ≠ {} ∧ mCaps = {} then return {}
4:      mReqs = MATCHREQS(M_R, {}, Reqs(N), Reqs(T), {})
5:      if Reqs(N) ≠ {} ∧ mReqs = {} then return {}
6:      mProps = MATCHPROPS(M_P, {}, Props(N), Props(T), {})
7:      if Props(N) ≠ {} ∧ mProps = {} then return {}
8:      mOps = MATCHOPS(M_O, {}, Ops(N), Ops(T), {})
9:      if Ops(N) ≠ {} ∧ mOps = {} then return {}
10:     candidates = {}
11:     mappings = mCaps × mReqs × mProps × mOps
12:     for all m ∈ mappings do
13:         C = COLOUR(T, m)
14:         if ∃⟨T, C', mappings_C'⟩ ∈ candidates : C = C' then
15:             mappings_C' = mappings_C' ∪ {m}
16:         else
17:             candidates = candidates ∪ {⟨T, C, {m}⟩}
18:     return candidates
```

FIGURE 4.4: MATCHMAKE function.

Sect. 3.1). The detected capability mappings are stored in $mCaps$ (line 2). Afterwards, MATCHMAKE checks whether all the required capabilities have been matched. If not, it ends by returning the empty set, which means that no fragments of T can match the target N (line 3). Analogously, the functions MATCHREQS, MATCHPROPS, and MATCHOPS are employed to determine $mReqs$, $mProps$, and $mOps$, respectively (lines 4-9). Once the sets of potential mappings are available, MATCHMAKE starts computing the set of candidate topology fragments (line 10). First, all the possible combinations of $mappings$ are created (line 11). Then, for each mapping m, COLOUR[4] determines the fragment C of T which exposes the features referred in m, and the candidate $\langle T, C, .\rangle$ is added or updated in the set of $candidates$ (lines 12-17). Finally, the set of $candidates$ is returned (line 18).

As illustrated above, the MATCHMAKE function employs the MATCHCAPS, MATCHREQS, MATCHPROPS, and MATCHOPS to detect the subsets of available capabilities, requirements, properties and operations which match the set of desired ones. MATCHCAPS (Fig. 4.5) is a recursive function which inputs the parameters M, $matched$, $needed$, and $available$. M is the matchmaking operator to be employed when comparing available capabilities with respect to the needed ones (e.g., $\equiv_C$, $\simeq_C$ in Sect. 3.1), while $matched$, $needed$, $available$, and $solutions$ are the parameters used to maintain the state of the recursive computation (namely, $matched$ contains the set of matchings detected by the current instance of MATCHCAPS, $needed$ is

[4]Due to its straightforward behaviour, we omit the presentation of COLOUR. Essentially, it "colours" the elements of the topology which offer the matched features, and returns the set of coloured elements.

```
1:  function MATCHCAPS(M, matched, needed, available)
2:      if ∀c_N ∈ needed, required(c_N) = 0 then return {matched}
3:      if available = {} then return {}
4:      select c_A from available
5:      available' = available − {c_A}
6:      solutions = MATCHCAPS(M, matched, needed, available')
7:      if ∃ c_N ∈ needed : (c_A M c_N ∧ required(c_N) > 0) then
8:          needed' = needed − {c_N}
9:          c_N' = c_N
10:         required(c_N') = required(c_N) − 1
11:         if required(c_N') > 0 then
12:             needed' = needed' ∪ {c_N'}
13:         matched' = matched ∪ {(c_N, c_A)}
14:         solutions' = MATCHCAPS(M, matched', needed', available')
15:         solutions = solutions ∪ solutions'
16:     return solutions
```

FIGURE 4.5: MATCHCAPS function.

the set of capability definitions which still need to be matched, and $available$ is the set of available capabilities). MATCHCAPS starts by checking whether there are no more $required$[5] capabilities in $needed$. If so, it returns $matched$ since it contains a potential mapping between available and desired capabilities (line 2). It then checks whether there are no more $available$ capabilities, which means that no mapping can be detected (line 3). If not, a capability c_A is removed from $available$ (line 4-5), and the $solutions$ without mappings to c_A are computed (line 6). Then, if c_A matches a needed capability c_N, a new instance of MATCHCAPS (with the sets $matched$ and $needed$ properly updated) is started so as to determine the solutions which comprise the mapping between c_N and c_A (lines 7-14). The computed $solutions'$ are then incorporated in the set $solutions$ determined by the current instance (line 15). Finally, the set of computed $solutions$ is returned (line 16). The functions MATCHREQS, MATCHPROPS, and MATCHOPS are analogous.

We are now able to matchmake N with respect to a single topology T of our repository $Repo$. In order to matchmake N with the entire repository, we just have to iteratively apply MATCHMAKE to all the topologies $T_i \in Repo$ and to unify the discovered candidates (Fig. 4.6).

For instance, by iterating the MATCHMAKE algorithm over the repository of applications in Fig. 4.3, we end up with the candidates illustrated in Fig. 4.7. The candidate (a) is composed by the software components of the *Moodle* application that offer the desired features, namely *ApachePHP-Module*, *ApacheWebServer* and *MySQLDBMS*, while (b) is composed only by the *XAMPP* server belonging to the *Wiki* application, since it offers all

[5] $required$ is defined as follows: If not explicitly assigned (as in line 10), $required(x)$ returns a default value. Such value is $x.lowerBound$ when x is a capabililty/requirement definition. Otherwise, it is 1.

1: **function** CANDIDATESUNION($N, Repo, M_C, M_R, M_P, M_O$)
2: $candidates = \{\}$
3: **for all** $T_i \in Repo$ **do**
4: $newCandidates =$ MATCHMAKE$(N, T, M_C, M_R, M_P, M_O)$
5: $candidates = candidates \cup newCandidates$
6: **return** $candidates$

FIGURE 4.6: CANDIDATESUNION function.

the desired features in a more integrated fashion. On the other hand, the topology implementing the *SendSMS* web service is removed from consideration, since MATCHMAKE fails at the very beginning (because none of the nodes appearing in the topology of the *SendSMS* application is offering the required capabilities).

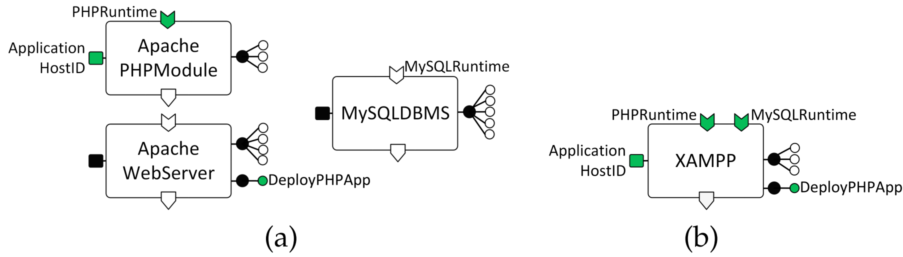


FIGURE 4.7: $Candidates$ determined for the example topologies in Fig. 4.3.

4.2.4 Election of the "best" candidate(s)

The number of detected candidates may be huge, and providing the user with all these candidates is not appropriate. As shown in Fig. 4.2, we reduce them by employing the steps RATE, FILTER and CUT, which are presented in this section.

RATE (Fig. 4.8) inputs a set of $candidates$ and a rating function r. It then constructs and outputs the set of $ratedCandidates$ by applying r to each of the candidate topology fragments (lines 2-6). Please note that we do not prescribe which rating function r to employ, since this depends on what the user wants to privilege. For instance, in our reference example we may look for the most integrated solutions, i.e. we may try to minimise the amount

1: **function** RATE($candidates, r$)
2: $ratedCandidates = \{\}$
3: **for all** $\langle T, C, mappings_C \rangle \in candidates$ **do**
4: $r_C = r(C, mappings_C, candidates)$
5: $ratedCandidates = ratedCandidates \cup \{\langle T, C, r_C, mappings_C \rangle\}$
6: **return** $ratedCandidates$

FIGURE 4.8: RATE function.

of components appearing in a candidate:

$$r(C) = 1/|C|$$

(where C is a candidate, and $|C|$ is the number of components it contains)[6]. The candidates (a) and (b) in Fig. 4.7 will then be rated $\frac{1}{3}$ and 1, respectively. TOSCAMART will thus privilege (b) with respect to (a).

Once the ratings are available, we can remove the "duplicates", namely the candidates having the same topology fragment C, the same rating r_C, and the same set of possible mappings $mappings_C$, independently from the topology T they come from. Please note, that both r_C and $mappings_C$ depend on the candidate topology fragment $elems$, since the former is a function of C, and the latter is the set of possible mappings between the features of N and those of the elements in C. Thus, we can consider duplicates those candidates having the same topology fragment C. This means that, in order to remove the duplicates from the output of RATE (by also optimising the performances of the method), we can merge RATE and FILTER into RATEANDFILTER (Fig. 4.9), so as to add candidates to the output only if they are not already there (lines 4-6). The function is also modified so that its output is a list (instead of a set — line 2), whose elements are descendingly sorted according to r_C (line 5).

1: **function** RATEANDFILTER($candidates, r$)
2: $ratedCandidates = [\,]$
3: **for all** $\langle T, C, mappings_C\rangle \in candidates$ **do**
4: **if** $\nexists\langle T', C', r_{C'}, mappings_{C'}\rangle \in ratedCandidates \ : \ C = C')$ **then**
5: $r_C = r(C, mappings_C, candidates)$
6: $\mathbf{add}_{sorted}\ \langle T, C, r_C, mappings_C\rangle$ **to** $ratedCandidates$
7: **return** $ratedCandidates$

FIGURE 4.9: RATEANDFILTER function.

Finally, CUT is implemented by cutting the list $ratedCandidates$ so as to maintain only the first Θ elements (Fig. 4.10). The value of Θ depends on the usage context. For a fully-automated approach, $\Theta = 1$ instructs TOSCAMART to proceed with the highest rated candidate. For instance, with $\Theta = 1$, our reference example (Fig. 4.7) proceeds by electing (b) as the candidate to be adapted (since its rating is 1, while (a) has a rating of $\frac{1}{3}$).

[6] There are many other possible rating functions. For instance, r could privilege not only the fragments having fewest components, but also the most frequent ones (by exploiting the amount of duplicates a candidate has).

$$r(C, candidates) = 1/|C| + \frac{duplicates(C, candidates)}{|candidates|},$$

where $duplicates(C, candidates)$ computes the number of duplicates of C among all candidate topology fragments in $candidates$.

```
1: function CUT(ratedCandidates, Θ)
2:     for i = |ratedCandidates| to Θ + 1 do
3:         remove ratedCandidates[i] from ratedCandidates
4:     return ratedCandidates
```

FIGURE 4.10: CUT function.

4.2.5 Adaptation of the elected candidate(s)

The topology fragments stored in $ElectedCandidates$ (i.e., those returned by the CUT function) have to be adapted so as to become concrete implementations of the target node type N. This is the purpose of the MAPPINGSELECTION and ADAPTATION steps.

For each candidate $\langle T, C, r_C, mappings_C \rangle$ in $ElectedCandidates$, MAPPINGSELECTION determines the mapping $m_C \in mappings_C$, that makes the candidate work as implementation of N. Despite there is no chance to ensure that the selected mapping is the one the user desires, we can approach the problem in a heuristic way, by selecting the mapping m_C which maps each feature to the "uppermost" available and compatible one. As a result, each candidate $\langle T, C, r_C, mappings_C \rangle$ in $ElectedCandidates$ is transformed into $\langle T, C, r_C, m_C \rangle$, where m_C is the selected mapping to be employed.

The selected mapping is then employed by the ADAPTATION function to transform each of the available topology fragments in a standalone implementation of N. First, the unsatisfied dependencies of the application components in the candidate fragment C are resolved. This is done by applying the following rules until C is no more modified by their operation:

A1) For each application component in C, its outgoing relationships must be added to C, if not already present. This rule does not affect the outgoing relationships whose sources are requirements that have been matched with those of the target node N.

A2) For each relationship in C, the components representing its source and target must be added to C, if not already present.

Once all (unsatisfied) dependencies have been processed, the actual adaptation can take place. The adaptation is analogous to the one we proposed in Sects. 3.1 and 3.2. Namely, (i) we create a new service template $newS$ which contains the application components and relationships stored in the topology fragment C, (ii) we define the boundary definitions of $newS$ by exposing only the features declared by N, and (iii) we employ m_C to map these features to the corresponding ones exhibited by the elements in the topology fragment. In this way, (from the topology fragment C), we build a new service template $newS$ that *exactly* matches the desired node type N (i.e., $newS \equiv N$), and that can thus be employed to concretely implement and substitute N.

In our reference example, TOSCAMART follows this approach to adapt the candidate fragment in Fig. 4.7(b). Namely, MAPPINGSELECTION selects the only available mapping and ADAPTATION starts extending the fragment according to A1 and A2. First, A1 causes the introduction of the *XAMPP* node's outgoing relationships. Then, A2 causes the introduction the *DebianOS* operating system component. Similarly, ADAPTATION introduces the relationship starting from *DebianOS*, as well as the *AmazonEC2* virtual machine, and this makes the candidate fragment be no more modifiable by A1 and A2. ADAPTATION then employs this fragment as the topology of a new service template, say *DesiredEnvironmentImplementation*. It then defines the boundaries of the new service template according to the selected mapping (Fig. 4.11). As a result, *DesiredEnvironmentImplementation* exactly matches the *DesiredEnvironment* target node type (Fig. 4.1), thus being a valid implementation for such a node type [33].

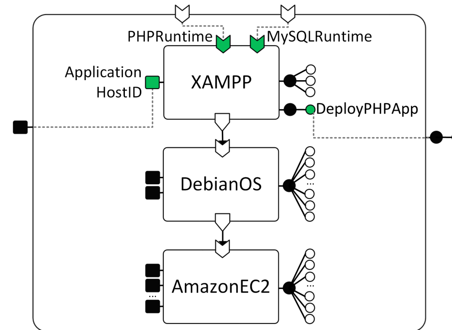


FIGURE 4.11: Implementation derived by TOSCAMART.

4.2.6 Orchestrating TOSCAMART

All the aforementioned functions must be orchestrated so as to operate the TOSCAMART method illustrated in Fig. 4.2. This can be easily done by implementing the function in Fig. 4.12. First, we invoke CANDIDATESUNION to derive all the *candidates* which can be excerpted from the topologies in *Repo* (line 2). These *candidates* are rated and filtered by employing the RATEANDFILTER function (line 3). The *filteredCandidates* are then reduced by CUT to the "best" Θ ones (line 4), whose mapping is subsequently determined by MAPPINGSELECTION (line 5). Finally, the resulting *mappedCandidates* are given to ADAPTATION so as to generate the *reusableImplementations* to be returned by TOSCAMART (lines 6-7).

4.3 Properties of TOSCAMART

In this section we discuss the termination, soundness, and time complexity of TOSCAMART. We do not discuss completeness, because it cannot be ensured on premise (as it depends on the function for rating candidates and

```
1: function TOSCAMART(N, T, r, Θ, M_C, M_R, M_P, M_O)
2:     candidates = CANDIDATESUNION(N, T, M_C, M_R, M_P, M_O)
3:     filteredCandidates = RATEANDFILTER(candidates, r)
4:     electedCandidates = CUT(filteredCandidates, Θ)
5:     mappedCandidates = MAPPINGSELECTION(electedCandidates)
6:     reusableImplementations = ADAPTATION(mappedCandidates)
7:     return reusableImplementations
```

FIGURE 4.12: TOSCAMART function.

on the heuristics for selecting a mapping for each candidate). In the following, please remember that the definitions of the topologies $T_i \in Repo$, as well as that of the target node type N, are necessarily finite.

Termination

Proposition 4.1 (Termination of TOSCAMART)**.** *The* TOSCAMART *procedure always* terminates.

Proof. According to Fig. 4.12, the termination of TOSCAMART directly follows from that of its steps. First, we need to ensure that MATCHMAKE (Fig. 4.4) eventually terminates. Consider MATCHCAPS (Fig. 4.5), which recurs on the set of $available$ capabilities until it becomes empty. Each recursive invocation is performed after the removal of a capability from the set $available$. Thus, since $available$ is initially finite, it eventually becomes empty, causing the termination of MATCHCAPS (which returns a finite set of $solutions$). Since the same holds for MATCHREQS, MATCHPROPS, and MATCHOPS, to prove the termination of MATCHMAKE, we just need to ensure that lines 10-17 eventually terminate. The set $mappings$ is computed as the product of finite sets, and thus both its computation and cardinality are finite. This, along with the fact that COLOUR can at most "colour" the whole finite topology, implies that the loop at lines 12-17 eventually terminates. Thus, MATCHMAKE eventually terminates.

The termination of CANDIDATESUNION, RATEANDFILTER, and CUT (Figs. 4.6, 4.9, and 4.10) obviously follows from the fact that MATCHMAKE produces a finite set of $Candidates$. Thus, we only have to prove the termination of MAPPINGSELECTION and ADAPTATION. MAPPINGSELECTION selects one of the potential mappings, for each of the candidates. Since the set of candidates and those containing their mappings are finite, we can conclude that MAPPINGSELECTION eventually terminates. This (along with the fact that the generation of the adapted service templates obviously terminates) implies that to prove the termination of ADAPTATION we just need to ensure that A1 and A2 eventually become no more applicable. This

can happen only if the size of the fragment can eventually be no more increased by their operation, which is true because the fragment is upper-bounded by the (finite) topology it comes from. It follows the termination of ADAPTATION, and thus that of the whole TOSCAMART approach. □

Soundness

We want to ensure that TOSCAMART returns (at most) Θ standalone service templates, which exactly match the target node type N (see Sect. 3.1), by properly adapting the Θ candidate fragments having the highest ratings.

Proposition 4.2 (Soundness of TOSCAMART)**.** *TOSCAMART is* sound*, i.e., it always returns (at most) Θ standalone service templates that exactly match the target node type N.*

Proof. First, we have to ensure that CANDIDATESUNION (Fig. 4.6) computes all possible candidate fragments (with all possible mappings) which can be excerpted from the cloud application topologies in *Repo*. This directly follows from the fact that MATCHMAKE computes all the possible mappings (and thus all the possible candidates) which can be derived from a single topology. Suppose (by contradiction) that MATCHMAKE misses one of these mappings. This can happen only if (at least) one among the procedures MATCHCAPS, MATCHREQS, MATCHPROPS, and MATCHOPS misses a mapping between a needed and an available feature. Suppose (without loss of generality) that MATCHCAPS misses a mapping between a needed capability definition c_N and a matching available capability c_A. This can happen only if the pair (c_A, c_N) is never added to a *matched* set, which in turns requires c_A to be never analysed (otherwise, since c_A matches c_N, line 13 would add (c_A, c_N) to a set of *matched* pairs). Nevertheless, according to the recursive definition of MATCHCAPS (Fig. 4.5), c_A is eventually analysed, and thus we come to a contradiction which allows us to deduce what we wanted to prove.

Then, we have to ensure that RATEANDFILTER and CUT remove the duplicates and reduce the set of available candidates to the Θ highest rated ones. This can be easily deduced from their definition (Figs. 4.9 and 4.10). RATEANDFILTER avoids the insertion of duplicate candidates through the check at line 4 and outputs the list of *ratedCandidates* sorted in descending order, according to r. Afterwards, CUT removes all the candidates whose index is higher than Θ, thus maintaining only the Θ candidates having the highest ratings.

For each of the Θ candidates, MAPPINGSELECTION selects one of the available mappings. This step thus provides ADAPTATION with the Θ candidates with the highest value of r, each containing only one mapping.

Finally, ADAPTATION has to ensure that each of the Θ candidates is transformed into a service template which (a_1) is standalone and (a_2) exactly matches N. (a_1) means that all the dependencies of the elements composing a candidate are satisfied, which can be easily proven by relying on the eventual non-applicability of the rules A1 and A2. (a_2) is ensured since the boundaries of each returned service template are built by including all the features declared by N (and by employing the selected mapping to map such features onto the internal ones which have been matched). Since the output of ADAPTATION coincides with that of TOSCAMART, it follows what we wanted to prove. □

Time complexity

Proposition 4.3 (Worst-case time complexity of TOSCAMART)**.** *Let r be the amount of application topologies available in the repository $Repo$, and let t be maximum amount of features that are exposed by an application topology in the repository $Repo$. The* worst-case time complexity *of* TOSCAMART *is:*

$$T(\text{TOSCAMART}) = O(r2^t).$$

Proof. Since TOSCAMART is a sequence of steps (Fig. 4.12), its (worst case) time complexity is given by the maximum among the (worst case) complexities of its steps.

Consider MATCHCAPS, and let $a = |available|$ and $n = |needed|$. In the base case, MATCHCAPS goes through the set of *needed* features, and thus its complexity can be approximated with $O(n)$. Otherwise, its complexity is dominated by the two recursive calls (whose a is decreased by 1) and by the union of the disjoint sets $solutions$ and $solutions'$. This, along with the fact that the size of $solutions$ and $solutions'$ can be upper-bounded by the size 2^{an} of the power set of the cartesian product $available \times needed$, allows us to derive the following recurrence relation[7]:

$$T(a) = \begin{cases} O(n) & \text{if } a = 0 \\ 2T(a-1) + O(an) & \text{if } a > 0 \end{cases}$$

By induction on a, it is possible to prove that the solution of the above relation is $T(a) = O(2^a n)$. This, along with the fact that initially $a = \max_{T \in Repo} |\mathsf{Caps}(T)|$ and $n = \mathsf{Caps}(N)$, allows us to conclude that

$$T(\text{MATCHCAPS}) = O(2^{\mathsf{Caps}(T)}\mathsf{Caps}(N)).$$

[7] According to Galil and Italiano [61], the union of two disjoint sets having size 2^s leads to a worst case complexity of $O(s)$.

Furthermore, since each recursive invocation having the set $available = \{\}$ can at most generate one mapping, we deduce that the maximum number of mappings can be $2^{\mathsf{Caps(T)}}$ (each of which contains $\sum_{x \in \mathsf{Caps}(N)} required(x)$ mappings). Similarly:

$$T(\text{MATCHREQS}) = O(2^{\mathsf{Reqs}(T)}\mathsf{Reqs}(N)),$$
$$T(\text{MATCHPROPS}) = O(2^{\mathsf{Props}(T)}\mathsf{Props}(N)),$$
$$T(\text{MATCHOPS}) = O(2^{\mathsf{Ops}(T)}\mathsf{Ops}(N)),$$

and the produced mappings can be upper-bounded with $2^{\mathsf{Reqs(T)}}$, $2^{\mathsf{Props(T)}}$, and $2^{\mathsf{Ops(T)}}$, respectively.

Consider now MATCHMAKE, which is dominated either by the matching procedures (lines 1-4) or by the generation of the *candidates* (lines 9-14). By properly combining the above computed quantities of mappings, we can conclude that the set $mappings$ can contain (at most) 2^t mappings, each consisting of m pairs, where:

$$t = |\mathsf{Caps}(T)| + |\mathsf{Reqs}(T)| + |\mathsf{Props}(T)| + |\mathsf{Ops}(T)|$$
$$m = \Sigma_{x \in \mathsf{Caps}(N) \cup \mathsf{Reqs}(N) \cup \mathsf{Props}(N) \cup \mathsf{Ops}(N)} required(x)$$

Thus, $candidates$ can be generated with a time complexity of $O(2^t m)$, which is higher than those of the matchmaking procedures. It follows that

$$T(\text{MATCHMAKE}) = O(2^t m).$$

However, in practice we have $m \ll t$, i.e., the amount m of features required by the target node type N is negligible with respect to the maximum amount t of features available in an application topology in the repository $Repo$. This allows us to approximate $T(\text{MATCHMAKE})$ as follows.

$$T(\text{MATCHMAKE}) = O(2^t).$$

From $T(\text{MATCHMAKE})$, we can deduce the complexity of CANDIDATES-UNION. Since the latter performs the union of disjoint sets, we can approximate the complexity of CANDIDATESUNION as that which comes out by operating MATCHMAKE against each topology in $Repo$, namely

$$T(\text{CANDIDATESUNION}) = O(r2^t).$$

where $r = |Repo|$. Furthermore, since all the remaining activities are dominated by set operations performed against $candidates$ (which can be viewed as a different representation of the above counted mappings), the steps from RATEANDFILTER afterwards can lead to a complexity which is at most

$O(r2^t)$. Thus, the worst-case complexity of TOSCAMART is

$$T(\text{TOSCAMART}) = O(r2^t).$$

□

4.4 Implementation

To illustrate the feasibility of TOSCAMART, we implemented a Java prototype[8] integrated into the open source *OpenTOSCA* ecosystem [15, 24, 78]. As shown in Fig. 4.13, our prototype processes TOSCA specifications taken

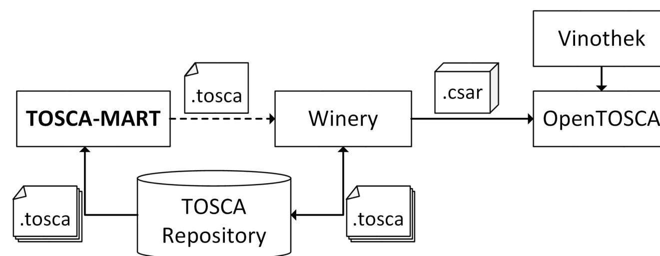


FIGURE 4.13: TOSCAMART in the OpenTOSCA ecosystem.

from a shared *TOSCA Repository*. It then produces new specifications which can then be imported into *Winery* [78], the graphical TOSCA modelling tool of the aforementioned open source ecosystem. Notice that the creation of new application specifications does suffice to enact the actual reuse of the matched fragments. Indeed, by employing Winery to replace the available specifications with the adapted ones, the existing TOSCA application packages (i.e., *Cloud Service ARchives*) can be processed as if they were only implementing the to-be-reused fragment. This is because each CSAR is processed by TOSCA containers (like OpenTOSCA [15]) according to the cloud application specification it contains [33, 95]. From the above, it follows that the adapted CSARs can also be provided to users through the Vinothek self-service portal [24].

We comparatively assessed the TOSCAMART prototype with respect to the matchmaking and adaptation approach we presented in Chapter 3. Notice that the latter is designed to match (and adapt) *one* available service template with *one* desired node type, and this is why we implemented a GREEDY prototype[9] of such approach that randomly access the shared repository

[8]The source code of the TOSCAMART prototype is publicly available on GitHub at https://github.com/jacopogiallo/TOSCA-MART/tree/master/TOSCA-MART.

[9]The source code of the GREEDY implementation is publicly available on GitHub at https://github.com/jacopogiallo/TOSCA-MART/tree/master/GreedyMatching.

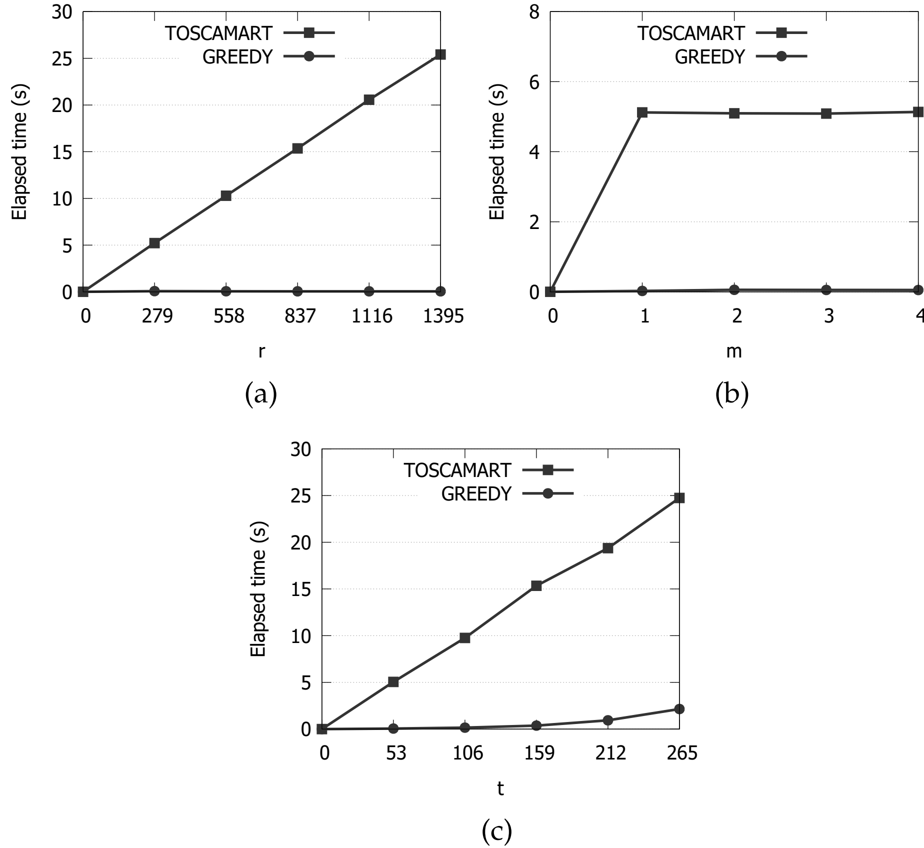


FIGURE 4.14: Time performances.

and returns (the adaptation of) the first[10] service template whose boundaries *plug-in* match the desired node type (see Sect. 3.1.2).

We then employed both prototypes to automatically generate an implementation of *DesiredEnvironment* (see Sect. 4.1) by relying on a plastic repository containing 279 validated TOSCA application topologies. Among them, 135 applications can offer the desired features, and only 27 can gain the highest rating.

We evaluated the time performances[11] of TOSCAMART and GREEDY with respect to the size r of the available repository, the maximum amount t of features available in a topology, and the amount m of features described in the desired node type (i.e., *DesiredEnvironment*). In order to test the prototypes under the same input conditions, we varied r by repeating multiple times the applications appearing in the repository, and t by making each application composed by multiple copies of its starting topology.

[10]We chose a GREEDY implementation for efficiency reasons (i.e., since the approach in Sects. 3.1 and 3.2 does not provide any way to rate matched service templates or to select the "best" ones, there is no reason to spend time in looking for all possible matches, but rather we can return the — adaptation of the — first match).

[11]All tests were repeated 300 times on a Windows 8.1 machine having an AMD A6-5400K APU (3.60GHz) and 4 GBs of RAM.

As expected, the completion time of TOSCAMART grew linearly with respect to growth of r (Fig. 4.14.(a)). The expectations were respected also when varying m, since the completion time was independent from m itself (Fig. 4.14.(b)). This is because we had $m \ll t$, and this allows us to approximate $T(\text{TOSCAMART})$ with $O(r2^t)$, which is independent from m (see Sect. 4.3). On the other hand, when we increased t, the completion time was not growing exponentially as expected. It instead grew linearly (Fig. 4.14.(c)). This is because, by properly implementing the matchmaking operators, the situations in which MATCHCAPS, MATCHREQS, MATCHPROPS, and MATCHOPS performed two recursive calls became negligible with respect to those in which they performed one recursive call. It follows that their complexity, as well as the number of detected mappings, can be approximated with $O(t)$, which in turn implies that $T(\text{TOSCAMART}) = O(rt)$. From the above, it follows that when (i) $m \ll t$ and (ii) the repository and matchmaking operators are such that the amount of matchings is negligible with respect to that of non-matchings, we have that the time complexity of TOSCAMART can be approximated with

$$T(\text{TOSCAMART}) = O(rt).$$

Please note that, in practice, these conditions are most probably true: (i) the features declared on a component are much fewer than those available in complex applications, and (ii) each complex application is composed by heterogeneous components offering different features, thus causing a negligible amount of matches among the performed checks — if the employed matchmaking operators are not dummy.

As shown in Fig. 4.14, we also compared the time performances of TOSCAMART with respect to those registered by the GREEDY (in the luckiest case of having all applications exposing all available features on their boundaries). As expected, TOSCAMART always required a completion time much higher than that of GREEDY, since the former always analyse all available applications, while the latter returns the first match. This is the price for providing the user with the topology fragments that best match the desired nodes, instead of providing the first match as a whole. However, it is worth noting that the development of complex application topologies is a process requiring days to be performed. Despite our solution requires a few seconds to complete, it allows cloud application developers to drastically reduce the time and effort they currently devote to the implementation of their cloud solutions.

Chapter 5

Modelling and analysing cloud application management

TOSCA provides a modelling language to describe, in a vendor-agnostic way, composite cloud applications and their management. Unfortunately, in its current version, TOSCA does not permit specifying the behaviour of the management operations of the components in a cloud application. More precisely, it is not possible to describe the order in which the operations of a component must be invoked, nor how those operations depend on the requirements or how they affect the capabilities of that component (and hence the requirements of other components they are connected to). This implies that the verification of whether a management plan is valid can only be performed manually, with a time-consuming and error-prone process.

In this chapter, after further motivating the need for a description of the behaviour of application components (see Sect. 5.1), we propose a simple extension of TOSCA that permits specifying the behaviour of management operations and their relations with states, requirements, and capabilities. More precisely, we define how to describe the management protocols of application components by means of finite state machines whose states and transitions are associated with conditions on the requirements and capabilities of components. Intuitively speaking, the objective of those conditions is to define the consistency of the states of a component, and to constrain the executability of the operations of a component to the satisfaction of its requirements (see Sect. 5.2).

We then show how the proposed extension permits automating various analyses of the management of complex cloud applications, like determining whether management plans are valid, which are their effects, or which plans permit reaching certain application configurations (see Sect. 5.3).

To illustrate the feasibilty of our approach, in Sect. 5.4 we describe a proof-of-concept, web-based application that permits editing the management protocols of TOSCA application components, and analysing plans that describe the management of a whole application. In Sect. 5.5 we also show how to exploit our approach to validate and automate the management of a concrete case study.

In Sect. 5.6 we also propose an alternative, Petri-net based semantics of *management protocols*, and we show how some of the analyses presented in Sect. 5.3 can be reduced to well-known problems in the Petri nets context.

5.1 Motivating scenario

Consider two utility web services, *Translator* and *Convertor*, and suppose that we want to manage them on a TOSCA-compliant cloud platform. After describing the services in TOSCA, we have to specify the third-party application components needed to properly host them. For instance, we may indicate that they have to run on an *Apache* server installed on a *Debian* operating system, which in turn runs on an *VMWare* virtual machine. Fig. 5.1 illustrates the resulting application topology. For the sake of readability, we focus only on the lifecycle interfaces [33] of each node in the topology (i.e., the interfaces containing the operations to install, configure, start, stop, and uninstall a component).

Suppose now that we want to specify the deployment of the *Translator* and *Convertor* services by writing a (workflow) plan. It is worth noting that, since TOSCA does not include any representation of the management protocols of (third-party) nodes, one may produce invalid plans. For instance, while Fig. 5.2 illustrates three seemingly valid BPMN plans, only (c) is a valid plan. Plan (a) is not valid since *Apache*'s *Configure* operation cannot be executed before *Apache* itself is running, while plan (b) is not valid since *Apache* cannot be installed if the *Debian* operating system is not running.

While the validity of management plans can be manually verified, this is a time-consuming and error-prone process. In order to enable the automated verification of the validity of plans, TOSCA needs to be extended with an explicit, machine-readable representation of the management protocols of the nodes in an application topology.

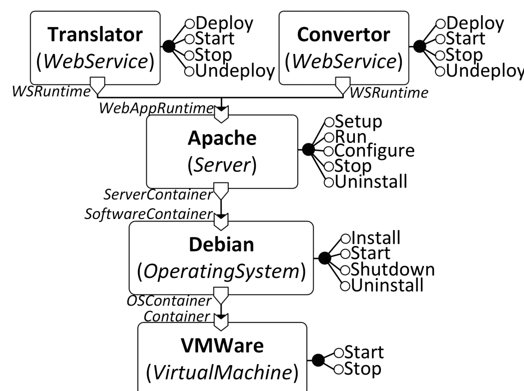


FIGURE 5.1: Motivating scenario.

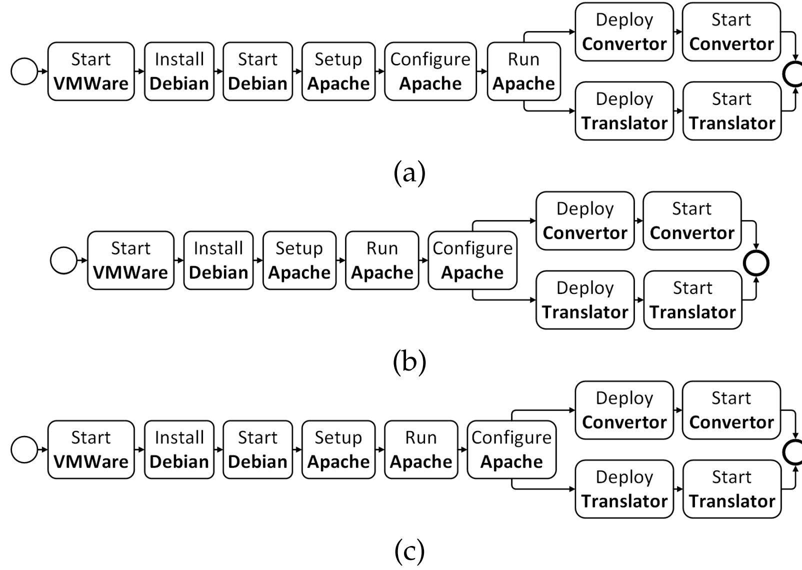


FIGURE 5.2: Examples of deployment plans.

5.2 Management protocols for cloud applications

TOSCA node types can be described by means of their states, requirements, capabilities, and management operations, but there is currently no way to specify how management operations affect states, how operations or states depend on requirements, or which capabilities are concretely provided in a certain state.

We hereby propose an extension of TOSCA that permits specifying the behaviour of management operations and their relations with states requirement and capabilities.

5.2.1 Definition of management protocols

Let N be a TOSCA node type, and let us denote its states, requirements, capabilities, and management operations with S_N, R_N, C_N, and O_N, respectively.

We want to describe whether and how the management operations of N depend on (i) other operations of the same node and/or on (ii) operations of other nodes providing the capabilities that satisfy the requirements of N.

(i) The first kind of dependencies can be easily described by specifying the relationship between states and management operations of N. More precisely, to describe the order with which the operations of N can be executed, we introduce a transition relation τ specifying whether an operation o can be executed in a state s, and which state is reached by executing o in s.

(ii) The second kind of dependencies can be described by associating transitions and states with (possibly empty) sets of requirements to indicate that the corresponding capabilities are assumed to be provided. More precisely, the requirements associated with a transition t specify which are the capabilities that must be offered to allow the execution of t. The requirements associated with a state of a node type N specify which are the capabilities that must (continue to) be offered by other nodes in order for N to (continue to) work properly.

To complete the description[1], we also permit associating each state s of a node type N with the capabilities provided by N in s, and to explictly specify which capabilities are maintained during a transition[2].

Definition 5.1 (Management protocol). *Let $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$ be a node type, where S_N, R_N, C_N, and O_N are the finite sets of its states, requirements, capabilities, and management operations. $\mathcal{M}_N = \langle \overline{s}_N, \rho_N, \chi_N, \tau_N \rangle$ is the* management protocol *of N, where*

- *$\overline{s}_N \in S_N$ is the initial state,*
- *ρ_N is a function indicating, for each state $s \in S_N$, which conditions on requirements must hold (i.e., $\rho_N(s) \subseteq R_N$),*
- *χ_N is a function indicating which capabilities of N are concretely offered in a state $s \in S_N$ (i.e., $\chi_N(s) \subseteq C_N$), and*
- *$\tau_N \subseteq S_N \times 2^{R_N} \times 2^{C_N} \times O_N \times S_N$ is a set of quintuples modelling the transition relation (i.e., $\langle s, P, X, o, s' \rangle \in \tau_N$ denotes that in state s, and if condition P holds, o is executable and leads to state s' — by maintaining the capability in X during the transition).*

Example 5.1. The management protocols of the node types in our motivating scenario (see Sect. 5.1) are shown in Fig. 5.3, where $\mathcal{M}_{WS}$ is the management protocol for *WebServices*, $\mathcal{M}_S$ is that for *Server*, $\mathcal{M}_{OS}$ is the management protocol for *OperatingSystem*, and $\mathcal{M}_{VM}$ is the management protocol for *VirtualMachine*.

Consider for instance the management protocol $\mathcal{M}_S$ of the *Server* node type, typing a *Tomcat* server. Its states S_S are *Unavailable* (initial), *Stopped*, and *Working*, the only requirement in R_S is *ServerContainer*, the only capability in C_S is *WebAppRuntime*, its management operations O_S are *Setup*, *Uninstall*, *Run*, *Stop*, and *Configure*. States *Unavailable* and *Stopped* are not associated with any requirement or capability. State *Working* instead specifies that the capability corresponding to the *ServerContainer* requirement must be provided in order for *Server* to (continue to) work properly. State *Working* also specifies that *Server* provides the *WebAppRuntime* capability when in such state. Finally, all transitions bind their executability to

[1] A proposal of syntax for management protocols can be found in Brogi et al. [27].

[2] In this chapter we present a proper extension of our initial definition of management protocols [26, 28, 29], where we were assuming that all capabilities were maintained during a transition. The extension will be further justified by Def. 6.5 (Chapter 6), which shows why transitions need to predicate on capabilities.

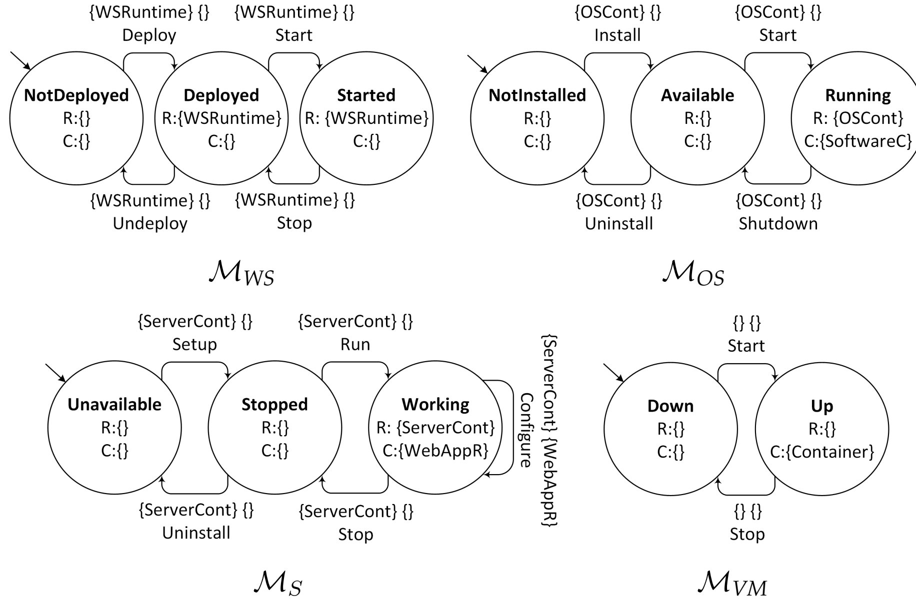


FIGURE 5.3: Management protocols of the node types in our motivating scenario.

the availability of the capability that satisfies the *ServerContainer* requirement. The transition involving *Configure* also specifies that it maintains the availability of the *WebAppRuntime* capability while being executed. □

5.2.2 Characterising management protocols

Management protocols (as per Def. 5.1) allow operations to have non-deterministic effects (e.g., a state may have two outgoing transitions corresponding to the same operation and leading to different states[3]). This form of non-determinism is not acceptable when managing TOSCA applications [33]. We will thus focus on *deterministic* management protocols (i.e., protocols ensuring deterministic effects when performing an operation in a state).

Definition 5.2 (Deterministic management protocols)**.** *Let* $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$ *be a node type, whose management protocol is* $\mathcal{M}_N = \langle \overline{s}_N, \rho_N, \chi_N, \tau_N \rangle$. $\mathcal{M}_N$ *is* deterministic *if and only if*

$$\forall \langle s_1, P_1, X_1, o_1, s_1' \rangle, \langle s_2, P_2, X_2, o_2, s_2' \rangle \in \tau_N : (s_1 = s_2 \wedge o_1 = o_2) \Rightarrow s_1' = s_2'$$

Furthermore, for each transition, the conditions on requirements should be coherent with the starting and target states. More precisely, the requirements assumed to hold in the starting state, as well as those assumed to hold in the target state, should also be assumed to hold during the transition, to avoid inconsistencies. Analogously, the capabilities that can be

[3]Note that the conditions of the two transitions may both hold even if the sets of requirements they refer to are disjoint. Hence the state obtained by performing the operation would be non-deterministic.

maintained during a transition are (at most) those offered by both its starting and target states.

Definition 5.3 (Well-formed management protocols)**.** *Let* $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$ *be a node type, whose management protocol is* $\mathcal{M}_N = \langle \overline{s}_N, \rho_N, \chi_N, \tau_N \rangle$. $\mathcal{M}_N$ *is* well-formed *iff*

$$\forall \langle s, P, X, o, s' \rangle \in \tau_N : \rho_N(s) \cup \rho_N(s') \subseteq P \ \wedge \ X \subseteq \chi_N(s) \cap \chi_N(s').$$

It is easy to check that all management protocols in Fig. 5.3 are deterministic and well-formed.

Example 5.2. An example of management protocols that is both non-deterministic and not well-formed is given in Fig. 5.4. Such protocol is non-deterministic since

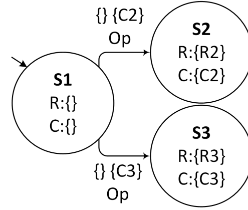


FIGURE 5.4: An example of management protocol that is non-deterministic and not well-formed.

S1 has two outgoing transition corresponding to operation *Op* and leading to two different states. It is also not well-formed since all transitions have conditions on requirements and capabilities that are not coherent with their source and target states. More precisely, both transitions do not assume the requirement assumed to hold in their target state, and they also maintain a capability which is not available in their source state. □

5.3 Analysis of management protocols

In this section, after describing how to infer the management behaviour of an application by composing the management protocols of its components, we describe different analyses that can be performed on such behaviour, such as checking the validity of a plan, determining its effects, or discovering plans that allow to reach certain application configurations.

5.3.1 Management behaviour of a composite application

We can now define the management behaviour of a composite cloud application by suitably composing the management protocols of the components that form such application.

Before digging into the details about the semantics of the management protocols in a composite cloud application, we introduce some shorthand

notation to denote generic composite applications, the nodes in their topology, and the connection among the requirements and capabilities of such nodes (e.g., to denote *Container* as the capability connected to the *OSContainer* requirement in our motivating scenario — Fig. 5.1).

Notation 5.1. *We denote with $A = \langle T, b \rangle$ a generic composite application, where T is the finite set of nodes in the application topology*[4]*, and where the connection between nodes is described by a (total)* binding *function*

$$b : \bigcup_{N \in T} R_N \rightarrow \bigcup_{N \in T} C_N$$

associating each node's requirement with the capability satisfying it.

Formally, the semantics of the management protocols in a composite application $A = \langle T, b \rangle$ can be defined by a labelled transition system[5] over configurations that denote the states of the nodes in T. Intuitively,

$$G \xrightarrow{o}_A G'$$

is a transition denoting that operation o can be executed (on a node) in A when the "global" state of A is G, making A evolve into the new global state G'. Hence, we first need to first formally define the notion of *global state* for a composite application.

Definition 5.4 (Global state of a composite application)**.** *Let $A = \langle T, b \rangle$ be a composite application, and let $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$. A* global state *$G$ of A is a set of states such that:*

$$G \subseteq \bigcup_{N \in T} S_N \wedge \forall N \in T : \exists! s \in G \cap S_N$$

We denote by $\overline{G}$ the initial global state A, where each node in T is in its initial state (viz., $\overline{G} = \bigcup_{N \in T} \overline{s}_N$).

We can now formally define the semantics of the management protocols in a composite application $A = \langle T, b \rangle$ (i.e., the *management behaviour* of A). Intuitively, a management operation o can be executed on a node $N \in T$ only if all the requirements needed by N to perform o are satisfied by the capabilities provided by (other) nodes in T.

Notation 5.2. *Let $A = \langle T, b \rangle$ be a composite application, and let $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle \in T$. To simplify notation, we shall denote with $\rho(G)$ the set of requirements that are assumed to hold by the nodes in T when A is in G, with*

[4] For simplicity, and without loss of generality, we assume that, given two nodes in a topology, the names of states, requirements, capabilities, and operations are all disjoint.

[5] An alternative, Petri net-based semantics of the management protocols in a composite application is presented in Sect. 5.6.

$\chi(G)$ the set of capabilities that are provided by such nodes in G, and with $b(R)$ the set of capabilities bound to the requirements in R. Formally:

- $\rho(G) = \bigcup_{N \in T}\{\rho_N(s) \mid s \in G \wedge s \in S_N\}$,
- $\chi(G) = \bigcup_{N \in T}\{\chi_N(s) \mid s \in G \wedge s \in S_N\}$, *and*
- $b(R) = \bigcup_{r \in R}\{b(r)\}$.

Definition 5.5 (Management behaviour of a composite application)**.** *Let $A = \langle T, b\rangle$ be a composite application, and let $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N\rangle$ with $\mathcal{M}_N = \langle \overline{s}_N, \rho_N, \chi_N, \tau_N\rangle$. The* management behaviour *of A is modelled by a labelled transition system whose configurations are the global states of A, and where the transition relation is defined by the following inference rule:*

$$\frac{s \in G \qquad \langle s, P, X, o, s'\rangle \in \tau_N \qquad b(P) \subseteq \chi(G)}{G \xrightarrow{o}_A (G - \{s\}) \cup \{s'\}}$$

Def. 5.5 permits modelling the evolution of a composite application A when a sequence of management operations is executed:

$$G_0 \xrightarrow{o_1}_A G_1 \xrightarrow{o_2}_A \cdots \xrightarrow{o_h}_A G_h.$$

5.3.2 Analysing the management of a composite application

While Def. 5.5 checks that the requirements needed by a node N to perform an operation o are satisfied by the capabilities provided by the (other) nodes in the topology of A, it does not check whether *after* performing o the requirements assumed by (the states of) all node templates continue to be satisfied (i.e., whether the capabilities satisfying them continue to be provided). We hence introduce the notion of *consistent* global state for a composite application.

Definition 5.6 (Consistency of a global state)**.** *Let $A = \langle T, b\rangle$ be a composite application, and let G be one of its global states. G is* consistent *if and only if*

$$b(\rho(G)) \subseteq \chi(G).$$

Defs. 5.5 and 5.6 allow us to formally characterise the *validity* of a sequence of management operations.

Definition 5.7 (Valid sequence of operations)**.** *Let $A = \langle T, b\rangle$ be a composite application. A sequence $o_1 o_2 \ldots o_n$ of management operations is* valid *from a global state G_0 of A if and only if:*

$$G_0 \xrightarrow{o_1}_A G_1 \xrightarrow{o_2}_A \cdots \xrightarrow{o_n}_A G_n$$

and each G_i is a consistent global state.

The validity of a *plan* (i.e., a workflow orchestrating the management operations of the nodes in a composite application) descends immediately from Def. 5.7.

Definition 5.8 (Valid plan)**.** *Let $A = \langle T, b \rangle$ be a composite application, and let G be one of its global states. A plan P for A is* valid *from G if and only if all its sequential traces are valid in G.*

Example 5.3. It is easy to see now that the deployment plan (c) of Fig. 5.2 is valid since, by starting from the initial global state, all its sequential traces are valid. Conversely, plans (a) and (b) in Fig. 5.2 are not valid as their traces are not valid. More precisely, plan (a) is not valid since all its sequential traces produce the derivation shown in Fig. 5.5, and *Apache*:*Configure* cannot be executed in the

VMWare	Debian	Apache	Translator	Convertor
Down	NotInstalled	Unavailable	NotDeployed	NotDeployed

↓ **VMare** : Start

VMWare	Debian	Apache	Translator	Convertor
Up	NotInstalled	Unavailable	NotDeployed	NotDeployed

↓ **Debian** : Install

VMWare	Debian	Apache	Translator	Convertor
Up	Available	Unavailable	NotDeployed	NotDeployed

↓ **Debian** : Start

VMWare	Debian	Apache	Translator	Convertor
Up	Running	Unavailable	NotDeployed	NotDeployed

↓ **Apache** : Setup

VMWare	Debian	Apache	Translator	Convertor
Up	Running	Stopped	NotDeployed	NotDeployed

FIGURE 5.5: Initial evolution according to plan (a) in Fig. 5.2.

reached global state (because it requires *Apache* to be in state *Working*, instead of *Stopped*).

On the other hand, plan (b) is not valid since all its traces start as shown in Fig. 5.6, and *Apache*:*Setup* cannot be executed in the reached global state. It indeed requires the capability satisfying *Apache*'s *ServerContainer* to be provided, but that capability is not provided when *Debian* is not in state *Running*. □

VMWare	Debian	Apache	Translator	Convertor
Down	NotInstalled	Unavailable	NotDeployed	NotDeployed

↓ **VMare** : Start

VMWare	Debian	Apache	Translator	Convertor
Up	NotInstalled	Unavailable	NotDeployed	NotDeployed

↓ **Debian** : Install

VMWare	Debian	Apache	Translator	Convertor
Up	Available	Unavailable	NotDeployed	NotDeployed

FIGURE 5.6: Initial evolution according to plan (b) in Fig. 5.2.

The introduced modelling can be exploited for various other purposes besides checking plans validity. For instance, validity of plans may not be enough, as their sequential traces may reach different global states. It is thus interesting to characterise deterministic plans.

Definition 5.9 (Deterministic plan)**.** *Let $A = \langle T, b\rangle$ be a composite application, and let G be one of its global states. A valid plan P for A is* deterministic *from G if and only if all its sequential traces reach the same global state G'.*

Remark 5.1. The way to check whether a given plan is valid or deterministic is obviously a visit of the graph associated with the transition system denoting the management behaviour of an application (Def. 5.5). Thanks to the constraints on management protocols and to the way they are combined, such a graph is *finite* and thus its visit is guaranteed to terminate. □

It is also interesting to compute the effects of a valid/deterministic plan P on the states of the components forming a composite application, as well as on the requirements that are satisfied and the capabilities that are available. Such effects can be directly determined from the global state(s) reached by performing the sequential traces of P. Moreover, the problem of finding whether there is a deployment plan which starts from the initial global state $\overline{G}$ and achieves a specific goal (e.g., bringing some components of an application to specific states or making some capabilities available) can be solved with a breadth-first search of the reachable global states. The same approach also works in the case of generic management plans (i.e., plans starting from a generic global state G), and it permits to find the sequential plans (if any) allowing to reach a certain goal from whatever starting G. It also allows to characterise an interesting property that a composite application may exhibit: If it is possible to reach the initial global state $\overline{G}$ from any G that is reachable from $\overline{G}$ itself, then it is always possible to generate a management plan for any (reachable) goal from any (reachable) global state. This ensures reversibility of actions, meaning that whatever G we reach from the initial global state $\overline{G}$, we can always get back to $\overline{G}$, thus always permitting a (soft) reset of the application.

Definition 5.10 (Soft-resettability)**.** *Let A be a composite application, and let $\overline{G}$ be its initial global state. We say that A is* softly resettable *if and only if for each global state G reached by executing a valid sequence of operations from $\overline{G}$, there exists a valid sequence of operations from G whose execution leads back to $\overline{G}$.*

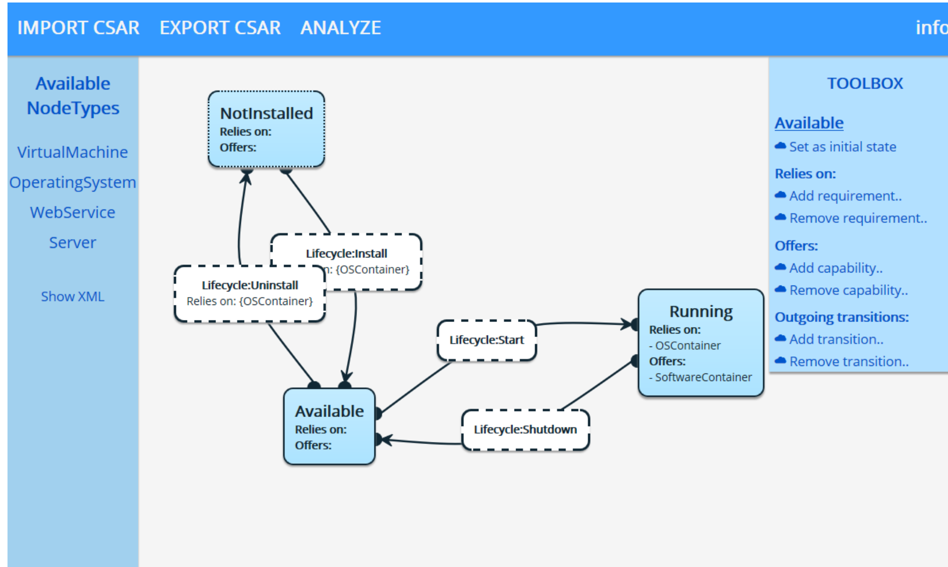
The above is a very convenient property, because it guarantees that it is always possible to generate a management plan for any reachable goal from any application state.

5.4 Implementation

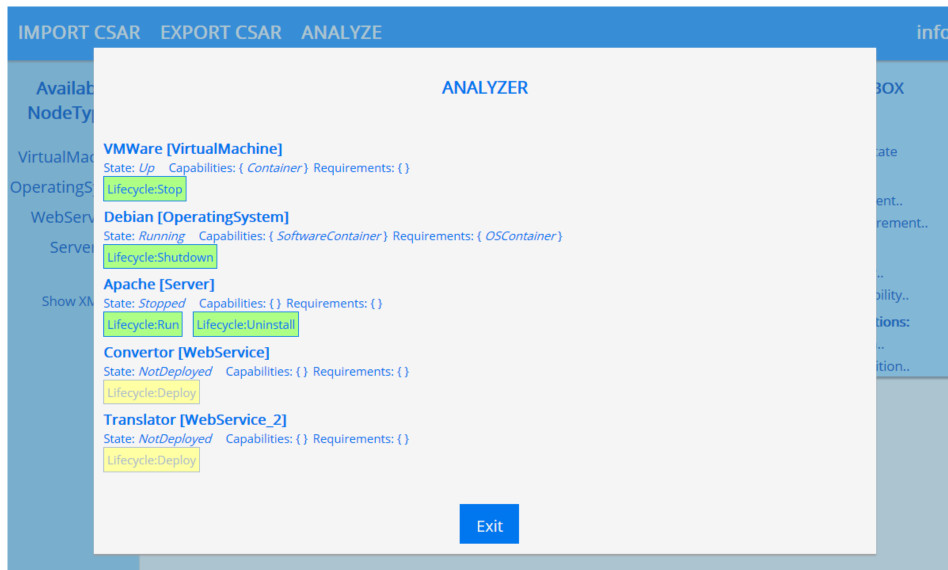
We now illustrate the feasibility of our approach by introducing BARREL, a web-based application[6] that permits editing and analysing management

[6]The application can be accessed at `http://ranma42.github.io/MProt/` with any modern web-browser, like Google Chrome or Mozilla Firefox. The source code is publicly available on GitHub at `https://github.com/ranma42/MProt`.

protocols in TOSCA applications. BARREL's interface is written in HTML5, while its back-end is written in TypeScript and JavaScript. In the following, we shall not deepen into implementation details, but rather focus on how BARREL can be used to edit and analyse existing TOSCA applications.



(a)



(b)

FIGURE 5.7: Screenshots of BARREL: (a) Editing mode, and (b) analysis mode.

The very first step is to import a CSAR package (see Sect. 2.2) containing a service template, as well as the node types instantiated in its topology. Once the CSAR is loaded, the names of the node types appear in the left hand pane of the interface of BARREL (*Available NodeTypes*), and by selecting one of them the user can start editing its management protocol (Fig. 5.7.(a)). The management protocol is visualised in the central pane, by displaying the states of the selected node type and the transitions among these states

(if any). By clicking on a state s, a dedicated *TOOLBOX* opens in the right pane. This *TOOLBOX* permits editing the current values of $\rho(s)$, $\chi(s)$, and $\tau(s)$, by allowing the user to update the set of requirements on which the selected state s relies, the set of capabilities it offers, and its outgoing transitions. Such updates can also be viewed directly in the XML source of the current node type, by clicking on the *Show XML* button in the left pane. Once the management protocols of all node types have been edited, the updated CSAR can be downloaded through the *EXPORT CSAR* functionality.

Users can also analyse the behaviour of the management operations appearing in the imported service template by selecting the *ANALYZE* option in the top menu. As a result, BARREL pops out a window showing the current global state of the application (Fig. 5.7.(b)). More precisely, the window lists all the node templates in the application topology, each associated with its current state, the requirements it relies on, the capabilities it offers and the operations actually available. Each operation is highlighted in green if all the capabilities connected to the requirements needed to execute it are currently available, otherwise it is highlighted in yellow. By clicking on a (green) operation users can simulate its execution, thus updating the current global state and then the *ANALYZER* window. If the reached state is inconsistent, a warning banner is displayed.

With the simple, interactive *ANALYZER* of BARREL, users can perform the analyses described in Sect. 5.3. For instance, to check whether a plan is valid, they just need to simulate its sequential traces and check that no inconsistent state is traversed. They can also compute the effects of a valid plan on states, capabilities and requirements by looking at the initial and final configurations displayed by the *ANALYZER* window. In this first version of BARREL, developers can only perform these analyses interactively, by manually clicking on the (green) operations and by looking at how they affect the global state.

It is worth noting that BARREL is already compatible with the OpenTOSCA open source ecosystem [15, 78]. BARREL is indeed able to process CSARs developed with the visual editor Winery [78], and it produces CSARs that can be imported both in Winery [78] and in OpenTOSCA [15]. Of course, while both Winery [78] and OpenTOSCA [15] import the CSARs generated by BARREL, they do not properly process the information concerning management protocols (since the extension to TOSCA we propose is not yet part of the TOSCA standard, and hence not yet supported in the OpenTOSCA open source environment).

5.5 Case study: *Thinking*

We hereby illustrate how management protocols (as well as BARREL) can be fruitfully exploited to analyse and orchestrate the management of a concrete case study[7].

5.5.1 The *Thinking* application

Thinking is an open-source web application that we implemented to permit end-users sharing what they are thinking about, so that all other users can read it. A snapshot of a running instance of the application is shown in Fig. 5.8.

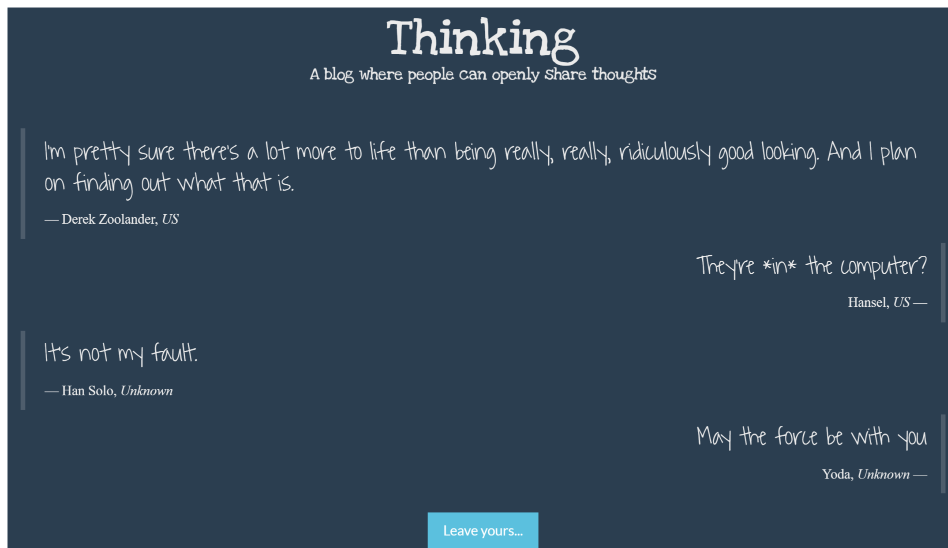


FIGURE 5.8: A snapshot of *Thinking*.

Thinking is composed by three main components: (i) an instance of MongoDB[8] that is exploited to permanently store the collection of thoughts shared by end-users, (ii) *ThoughtsApi*[9], which is a Dropwizard-based REST API that permits remotely accessing the collection of shared thoughts, and (iii) *ThoughtsGui*[10], which is a web-based graphical user interface that interacts with *ThoughtsApi* to permit retrieving and adding thoughts to the shared collection. The MongoDB instance is obtained by instantiating a *Mongo* Docker container, while *ThoughtsApi* and *ThoughtsGui* are made concrete by hosting them on a *Maven* Docker container and on a *Node* Docker

[7]The case study has been run on an Ubuntu 16.04 LTS virtual machine, with 32 GB of storage and 8 GB of memory.

[8]https://www.mongodb.com/.

[9]The source code of *ThoughtsApi* is publicly available on GitHub at https://github.com/jacopogiallo/thoughts-api.

[10]The source code of *ThoughtsGui* is publicly available on GitHub at https://github.com/jacopogiallo/thoughts-gui.

container, respectively[11]. The resulting application topology of *Thinking* is depicted in Fig. 5.9.

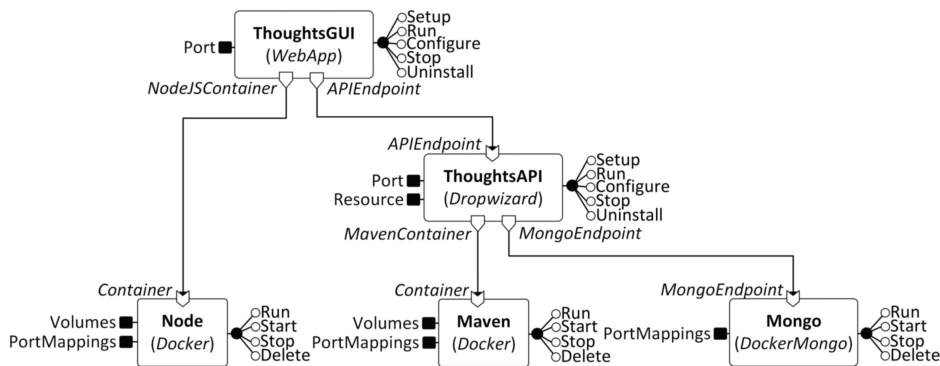


FIGURE 5.9: Topology of the application *Thinking*.

Mongo

Mongo offers a *MongoEndpoint* capability (that is used to satisfy the corresponding requirement of *ThoughtsApi*), and a property that permits specifying how to map the ports of the container onto the ports of the host (e.g., MongoDB is offering its functionalities on the port 27017 of the container, and to make such functionalities available outside of the container we may map such port to the same port on the host). It is also offering the operations to *Run*, *Start*, *Stop*, and *Delete* the container[12].

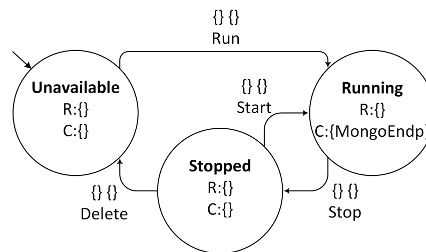


FIGURE 5.10: Management protocol for *Mongo*'s node type.

The management protocol of *Mongo* is shown in Fig. 5.10. Its initial state is *Unavailable*, where *Mongo* is not providing any capability, and where it can perform *Run* to become *Running*. In the *Running* state *Mongo* continues to provide its *MongoEndpoint* capability, thus satisfying the requirements connected to it. It also permits executing the *Stop* operation, which makes

[11]Further information about Docker and its fundamentals (e.g., port mappings, volumes) can be found at `https://www.docker.com/` or in the Docker user guide [51].

[12]The implementation of the management operations of *Mongo* is as follows. *Run* is implemented by the command line instruction "`docker run -name mongo -p 27017:27107 -d mongo`", *Start* by "`docker start mongo`", *Stop* by "`docker stop mongo`", and *Delete* by "`docker rm mongo`".

Mongo transit to the *Stopped* state, where it stops providing the *MongoEndpoint* capability. From the *Stopped* state, *Mongo* can return to be *Running* or *Unavailable*, respectively by executing the operations *Start* or *Delete*.

Maven and *Node*

Maven and *Node* are both of type *Docker*, and their structure is similar to that of *Mongo*: They offer a capability to satisfy the requirements of other components, a property for specifying the port mappings, and the operations to *Run*, *Start*, *Stop*, and *Delete* them[13]. It also provides a property for specifying the volumes to mount (e.g., *Maven* may mount a "*/thoughts-api/*" volume where to place all the sources of *ThoughtsApi*, and *Node* may mount a "*/thoughts-gui/*" volume where to place all the sources of *ThoughtsGui*).

As illustrated by Fig. 5.11, the management protocol of *Maven* and *Node* is also analogous to that of *Mongo*, with the only difference that *Maven* and *Node* continue to provide the *Container* capability in their *Running* state (instead of the *MongoEndpoint* capability offered by *Mongo*).

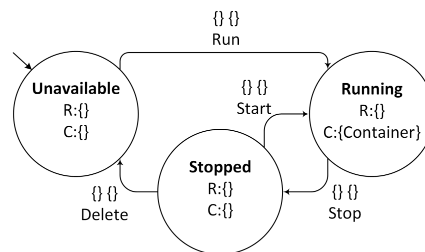


FIGURE 5.11: Management protocol for nodes of type *Docker*.

ThoughtsApi

ThoughtsApi is a *Dropwizard* application, which offers a REST API as a resource on a given port. Such information can be specified with dedicated properties. *ThoughtsApi* also offers an *APIEndpoint* capability (that is used to satisfy the corresponding requirement of *ThoughtsGui*), and the set of management operations implementing its lifecycle[14]. It also requires a *MavenContainer* where to be installed, and a *MongoEndpoint* where to connect.

The management behaviour of *ThoughtsApi* is displayed in Fig. 5.12. Its states are *Uninstalled* (initial), *Installed*, *Configured*, and *Working*. States

[13]The implementation of the management operations of *Maven* and *Node* is analogous to that of *Mongo*, with only some differences concerning the implementation of *Run*. Further details on how to implement *Run* can be found at https://github.com/jacopogiallo/thoughts-api (for *Maven*) and at https://github.com/jacopogiallo/thoughts-gui (for *Node*).

[14]The bash scripts implementing the operations to *Setup*, *Start*, *Configure*, *Stop*, and *Uninstall* *ThoughtsApi* are publicly available on GitHub at https://github.com/jacopogiallo/thoughts-api/tree/master/scripts.

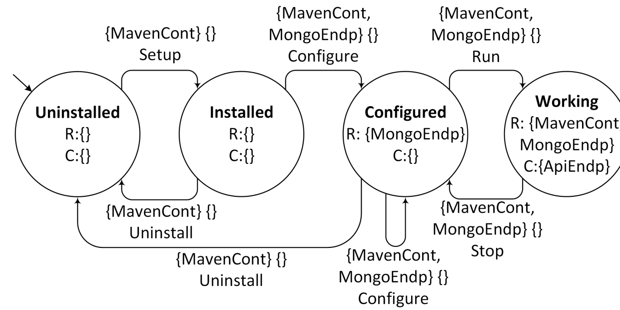


FIGURE 5.12: Management protocol for *ThoughtsApi*'s node type.

Uninstalled and *Installed* are not associated with any requirement or capability, while states *Configured* and *Working* specify that the capability corresponding to the *MongoEndpoint* requirement must continue to be provided in order for *ThoughtsApi* to continue to work properly. State *Working* also specifies that the capability corresponding to the *MavenContainer* requirement must continue to be provided, and that *ThoughtsApi* continues to provide the *APIEndpoint* capability when in such state. Finally, all transitions bind their executability to the availability of the capability that satisfies the *MavenContainer* requirement. *Configure*, *Run*, and *Stop* bind their executability also to the availability of the capability that satisfies *MongoEndpoint* requirement.

ThoughtsGui

ThoughtsGui is a web-application offering a web-based GUI for connecting to the *Thinking* application. *ThoughtsGui* can be reached at a given port, which can be specified through the dedicated property, and it offers the operations to manage its lifecycle[15]. To effectively run, it also require a container where to be installed and the endpoint of the API where to connect to (through its requirements *NodeJSContainer* and *APIEndpoint*, respectively).

The management protocol of *ThoughtsGui* is illustrated in Fig. 5.13. Its states are *Uninstalled* (initial), *Installed*, *Configured*, *Running*, and *Working*. States *Uninstalled* and *Installed* are not associated with any requirement or capability, while states *Running*, *Configured*, and *Working* specify that the capabilities corresponding to the indicated requirements must continue to be provided in order for *ThoughtsGui* to continue to work properly. All transitions bind their executability to the availability of the capability that satisfies the *NodeJSContainer* requirement. The transition targeting or outgoing from *Working* bind their executability also to the availability of the capability that satisfies *APIEndpoint* requirement.

[15]The bash scripts implementing the operations to *Setup*, *Start*, *Configure*, *Stop*, and *Uninstall* of *ThoughtsGui* are publicly available on GitHub at https://github.com/jacopogiallo/thoughts-gui/tree/master/scripts.

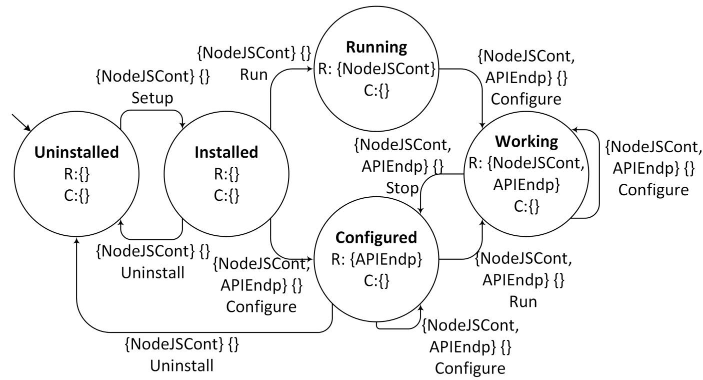


FIGURE 5.13: Management protocol for *ThoughtsGui*'s node type.

5.5.2 Analysing *Thinking*'s deployment plans

We hereby show how the analyses we described in Sect. 5.3 (as well as the BARREL tool — Sect. 5.4) can be exploited to validate deployment plans for *Thinking*. We also show how the execution of valid plan results in effectively deploying the *Thinking* application, and how the same does not hold for non-valid plans[16].

Consider the deployment plans in Fig. 5.14, which differ only for the order with which the *Start* and *Configure* operations of both *ThoughtsApi* and *ThoughtsGui* are invoked. Plan (a) is valid in the initial global state, since we

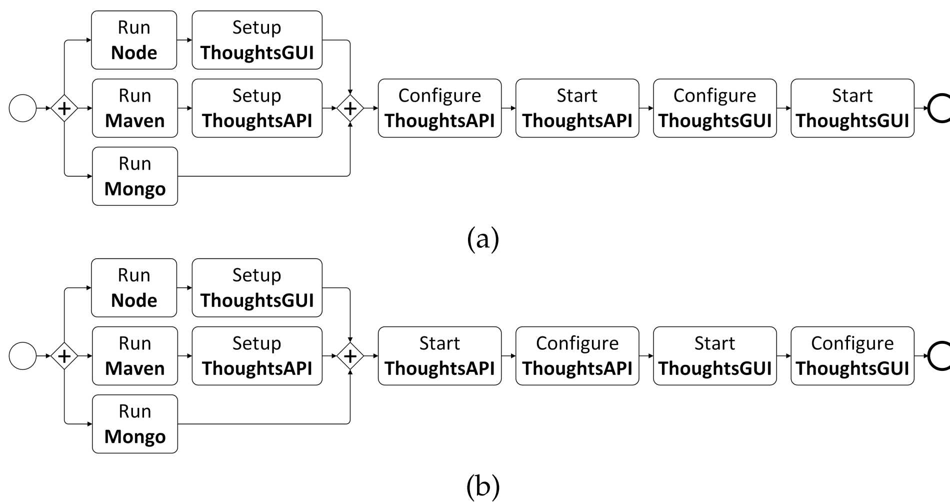


FIGURE 5.14: Two deployment plans for *Thinking*.

can easily verify that all its sequential traces are valid operation sequences in the initial global state. By executing any of such sequential traces, we end up with a "fresh" instance of *Thinking*, such as that in Fig. 5.15.

On the other hand, we can easily verify that plan (b) cannot be considered valid, since the management protocol of *ThoughtsApi* (Fig. 5.12) does

[16]Plans have been manually executed in a 64-bit Ubuntu 16.04 LTS virtual machine, with 32 GB of storage, and 8 GB of memory.

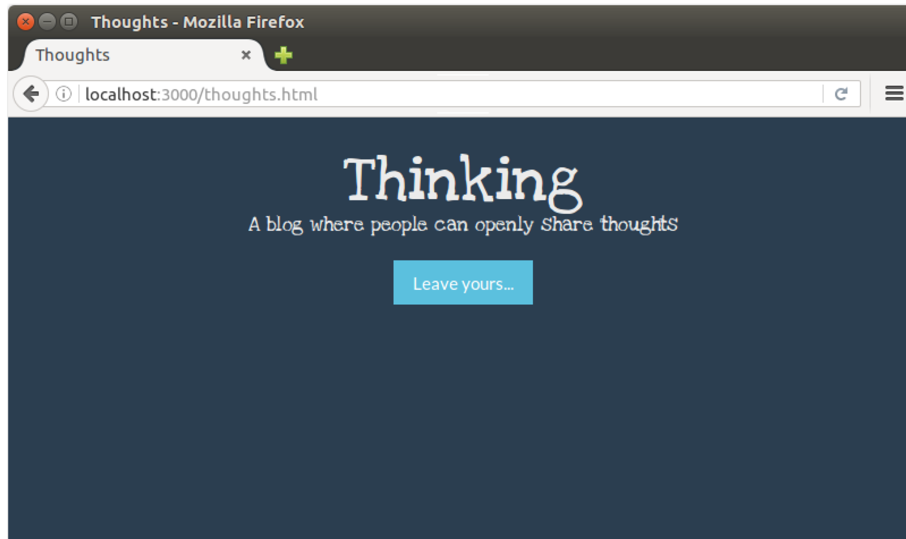


FIGURE 5.15: A "fresh" instance of *Thinking*.

not permit executing *Start* before *Configure*. The latter is because *ThoughtsApi* can be started only after having set the active endpoint of the MongoDB storing the collection of thoughts. The execution of *Start* before *Configure* indeed results in raising the exception in Fig. 5.16, which in turns means that *ThoughtsApi* is not able to effectively serve its clients.

```
INFO  [2016-08-04 14:38:05,071] org.mongodb.driver.cluster: Exception in monitor
 thread while connecting to server unknown:27017
! java.net.UnknownHostException: unknown: unknown error
! at java.net.Inet6AddressImpl.lookupAllHostAddr(Native Method)
! at java.net.InetAddress$2.lookupAllHostAddr(InetAddress.java:928)
! at java.net.InetAddress.getAddressesFromNameService(InetAddress.java:1323)
! at java.net.InetAddress.getAllByName0(InetAddress.java:1276)
! at java.net.InetAddress.getAllByName(InetAddress.java:1192)
! at java.net.InetAddress.getAllByName(InetAddress.java:1126)
! at java.net.InetAddress.getByName(InetAddress.java:1076)
! at com.mongodb.ServerAddress.getSocketAddress(ServerAddress.java:186)
! ... 5 common frames omitted
! Causing: com.mongodb.MongoSocketException: unknown: unknown error
! at com.mongodb.ServerAddress.getSocketAddress(ServerAddress.java:188)
! at com.mongodb.connection.SocketStreamHelper.initialize(SocketStreamHelper.jav
a:50)
! at com.mongodb.connection.SocketStream.open(SocketStream.java:58)
! at com.mongodb.connection.InternalStreamConnection.open(InternalStreamConnecti
on.java:114)
! at com.mongodb.connection.DefaultServerMonitor$ServerMonitorRunnable.run(Defau
ltServerMonitor.java:128)
! at java.lang.Thread.run(Thread.java:745)
```

FIGURE 5.16: Exception raised by *ThoughtsApi* when started without being configured to connect to the endpoint of a MongoDB instance.

Consider now the plan in Fig. 5.17. It cannot be considered valid in the initial global state, since its sequential traces are not valid in the initial global state. For instance, some of such sequences try to configure *ThoughtsApi* before *Mongo* is up and running (i.e., they execute *ThoughtsApi*'s *Configure* before *Mongo*'s *Run*). Despite this may not cause real issues when trying to deploy *Thinking* on a local host (where endpoints can be decided statically), the same does not hold in environments where endpoints are

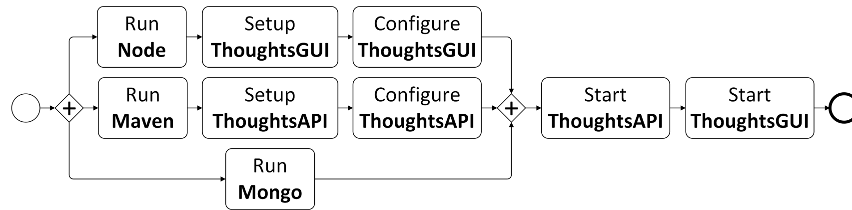


FIGURE 5.17: A deployment plan for *Thinking*.

assigned on-demand dynamically, such as the cloud. In a cloud-based deployment (e.g., Docker cloud[17]) we may not know which URL will be assigned to *Mongo* until the latter has been started, and this means that if we try to execute *ThoughtsApi*'s *Configure* before *Mongo*'s *Run* we may miss some configuration parameters.

5.5.3 Planning the undeployment of *Thinking*'s GUI and API

Consider the running instance of *Thinking* displayed in Fig. 5.15, and suppose that we wish to undeploy its components *ThoughtsGui* and *ThoughtsApi* (i.e., that we wish to come back to the global state where *Mongo* is the only component installed and running).

As we discussed in Sect. 5.3, the problem of finding whether there is a plan starting from a global state and reaching another global state can be solved with a breadth-first search of the reachable global states. If we apply this approach to our situation, we discover that one of the shortest, valid sequences of operations that permit undeploying the components *ThoughtsGui* and *ThoughtsApi* from a running instance of *Thinking* is that displayed in Fig. 5.18. By executing such sequence of operations, we effectively come back to the situation where no container but that of *Mongo* is installed and running (Fig. 5.19).

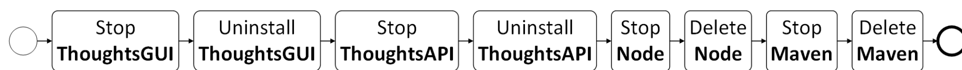


FIGURE 5.18: An undeployment plan for *Thinking*.

```
jacopo@yellow:~$ docker ps -a
CONTAINER ID   IMAGE  COMMAND                  CREATED         STATUS        PORTS
f385a566c619   mongo  "/entrypoint.sh mongo"  17 minutes ago Up 17 minutes 27017/tcp
jacopo@yellow:~$ 
```

FIGURE 5.19: Snapshot displaying the effective undeployment of the components *ThoughtsGui* and *ThoughtsApi* from a running instance of *Thinking*.

[17] https://cloud.docker.com.

5.6 Petri net-based analysis of management protocols

We hereby show how management protocols can be naturally modelled, in a compositional way, by means of open Petri nets. The proposed modelling permits automating various different analyses (such as determining whether a deployment plan is valid, which are its effects, or which plans allow to reach certain application configurations) in an alternative way with respect to that presented in Sect. 5.3.

5.6.1 Background: (Open) Petri nets

Before providing a formal definition of open Petri nets (Def. 5.12), we recall the definition of Petri nets just to introduce the employed notation. We instead omit to recall other very basic notions about Petri nets (e.g., marking of a net, firing of transitions, etc.) as they are well-know and easy to find in literature (e.g., in the work by Murata [89]).

Definition 5.11 (Petri net)**.** *A* Petri net *is a tuple* $\mathcal{P} = \langle P, T, \bullet\cdot, \cdot\bullet, M_0\rangle$ *where* P *is a set of places,* T *is a set of transitions (with* $P \cap T = \varnothing$*),* $\bullet\cdot, \cdot\bullet : T \to 2^P$ *are functions assigning to each transition its input and output places, and* $M_0 : P \to \mathbb{N}$ *is the initial marking of* $\mathcal{P}$*.*

According to Baldan et al. [9], an open Petri net is an ordinary Petri net with a distinguished set of (open) places that are intended to represent the interface of the net towards the external environment, meaning that the environment can put or remove tokens from those places. In this chapter, we shall employ a subset of open Petri nets, where transitions consume at most one token from each place, and where the environment can both add/remove tokens to/from all open places.

Definition 5.12 (Open Petri net)**.** *An* open Petri net *is a pair* $\mathcal{Z} = \langle \mathcal{P}, I\rangle$*, where* $\mathcal{P} = \langle P, T, \bullet\cdot, \cdot\bullet, M_0\rangle$ *is an ordinary Petri net, and* $I \subseteq P$ *is the set of* open *places. The places in* $P \setminus I$ *will be referred to as* internal *places.*

5.6.2 Encoding management protocols in Petri nets

A (deterministic) management protocol $\mathcal{M}_N = \langle \overline{s}_N, \rho_N, \chi_N, \tau_N\rangle$ of a node type N can be easily encoded by an open Petri net. Each state of $\mathcal{M}_N$ is mapped to an internal place of the Petri net, and each capability and requirement of N is mapped to an open place of the same net. Furthermore, each transition $\langle s, P, X, o, s'\rangle$ of $\mathcal{M}_N$ is mapped into a Petri net transition t with the following inputs and outputs:

(i) The input places of t are the places denoting s, the requirements that are needed but not already available in s (i.e., $(\rho_N(s') \cup P) - \rho_N(s)$), and the capabilities that are provided in s but not in s' (i.e., $\chi_N(s) - \chi_N(s')$).

(ii) The output places of t are the places denoting s', the requirements that were needed but are no more assumed to hold in s' (i.e., $(\rho_N(s) \cup P) - \rho_N(s')$), and the capabilities that are provided in s' but not in s (i.e., $\chi_N(s') - \chi_N(s)$).

The initial marking of the obtained net prescribes that the only place initially containing a token is that corresponding to the initial state $\overline{s}$ of $\mathcal{M}_N$.

Definition 5.13 (Encoding of management protocols). *Let $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$ be a node, with $\mathcal{M}_N = \langle \overline{s}_N, \rho_N, \chi_N, \tau_N \rangle$. The* management protocol $\mathcal{M}_N$ is encoded into an open Petri net $\mathcal{Z}_N = \langle \mathcal{P}_N, I_N \rangle$*, with $\mathcal{P}_N = \langle P_N, T_N, \bullet\cdot, \cdot\bullet, M_0 \rangle$ and $I_N \subseteq P_N$, as follows.*

- *$P_N = S_N \cup R_N \cup C_N$, i.e. the set P_N of places contains a separate place for each state in S_N, for each requirement in R_N, and for each capability in C_N.*
- *$I_N = R_N \cup C_N$, i.e. the set $I_N \subset P_N$ of open places contains the places denoting the requirements in R_N and the capabilities in C_N.*
- *$T_N = \tau_N$ (i.e., the set T_N contains a net transition t for each transition $\langle s, P, X, o, s' \rangle \in \tau_N$), and $\forall t = \langle s, P, X, o, s' \rangle \in T_N$:*
  - *(i) $\bullet t = \{s\} \cup ((\rho_N(s') \cup P) - \rho_N(s)) \cup (\chi_N(s) - \chi_N(s'))$, i.e. the set $\bullet t$ of input places contains the place s, the places denoting the requirements in $(\rho_N(s') \cup P) - \rho_N(s)$, and those denoting the capabilities in $\chi_N(s) - \chi_N(s')$.*
  - *(ii) $t\bullet = \{s'\} \cup ((\rho_N(s) \cup P) - \rho_N(s')) \cup (\chi_N(s') - \chi_N(s))$, i.e. the set $t\bullet$ of output places contains the place s', the places denoting the requirements in $(\rho_N(s) \cup P) - \rho_N(s')$, and those denoting the capabilities in $\chi_N(s') - \chi_N(s)$.*
- *The initial marking M_0 of $\mathcal{Z}_N$ is defined as follows:*

$$\forall p \in P_N . M_0(p) = \begin{cases} 1 & \text{if } p \text{ denotes } \overline{s}_N \\ 0 & \text{otherwise} \end{cases}$$

The above definition ensures that the Petri net encoding of a management protocol satisfies the following properties:

- There is a one-to-one correspondence between the marking of the internal places of the Petri net and the states of a management protocol. Namely, there is exactly one token in the internal place denoting the current state, and no tokens in the other internal places.
- Each operation can be performed if and only if all the necessary requirements are available in the source state, and no capability required by any connected component is disabled in the target state.

Example 5.4. Consider for instance the management protocol $\mathcal{M}_S$ (Fig. 5.3), whose corresponding Petri net is shown in Fig. 5.20. Each state in $\mathcal{M}_S$ is translated into

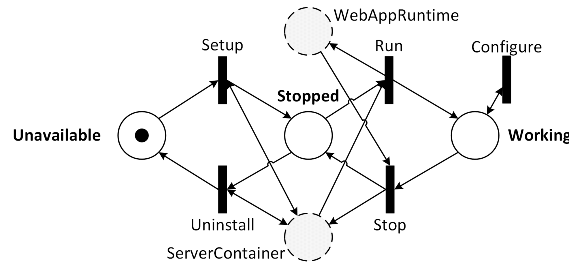


FIGURE 5.20: Example of Petri net translation.

an internal place (represented as a circle), while the *ServerContainer* requirement and the *WebAppRuntime* capability are translated into open places (represented as dashed circles). Additionally, protocol transitions are translated into net transitions. For example, the transition $\langle Stopped, \{ServerContainer\}, Run, Working\rangle$ is translated into a Petri net transition, whose inputs places are *Stopped* and *ServerContainer*, and whose outputs places are *Working* and *WebAppRuntime*. □

5.6.3 Modelling the management of a cloud application

We now show how the Petri net modelling the management protocol of a whole application can be obtained, in a compositional way, from the Petri nets modelling the management protocols of the nodes in its topology.

We first need to model (by open Petri nets working as a *capability controllers*) the relationships that define in a topology the association between the requirements of a node and the capabilities of other nodes. To do that, we first define an utility *binding* function that returns the set of requirements with which a capability is associated.

Definition 5.14 (Binding). *Let $A = \langle T, b\rangle$ be a composite application, and let c be a capability offered by a node in T. Let also Let $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N\rangle$. We define* binding$(c, A)$ *as follows:*

$$\text{binding}(c, A) = \bigcup_{N \in T} \{r \in R_N \mid b(r) = c\}.$$

We now exploit function *binding* to define *capability controllers*. On the one hand, the controller must ensure that once a capability c is available, the nodes exposing the connected requirements $r_1, \ldots, r_n$ are able to simultaneously exploit it. This is obtained by adding a transition $c_\uparrow$ able to propagate the token from place c to places $r_1, \ldots, r_n$ (i.e., the input place of $c_\uparrow$ is c, and its output places are $r_1, \ldots, r_n$). On the other hand, the controller has also to ensure that the capability is not removed while at least another node is

actively assuming its availability (with a condition on a connected requirement). Thus, we introduce a transition $c_\downarrow$ whose input places are $r_1, \dots, r_n$ and whose output place is c.

Definition 5.15 (Controller). *Let $A = \langle T, b \rangle$ be a composite application, and let c be a capability offered by a node in T. Let $r_1, \dots, r_n$ be the requirements exposed by the nodes in T such that* binding$(c, A) = \{r_1, \dots, r_n\}$. *The* controller *of c is an open Petri net $\mathcal{Z}_c = \langle \mathcal{P}_c, I_c \rangle$, with $\mathcal{P}_c = \langle P_c, T_c, \bullet\cdot, \cdot\bullet, M_0 \rangle$, defined as follows.*

- *The set P_c of places contains a separate place for the capability c and for each requirement $r_1, \dots, r_n$. It also contains a place r_c that witnesses the availability of the capability c.*
- *The set I_c coincides with P_c.*
- *The set T_c contains only two Petri net transitions $c_\uparrow$ and $c_\downarrow$.*
  - *The input place of $c_\uparrow$ is c (i.e., $\bullet c_\uparrow = \{c\}$). The output places of $c_\uparrow$ are $r_1, \dots, r_n$ and r_c (i.e., $c_\uparrow \bullet = \{r_1, \dots, r_n\} \cup \{r_c\}$).*
  - *The input places of $c_\downarrow$ are $r_1, \dots, r_n$ and r_c (i.e., $\bullet c_\uparrow = \{r_1, \dots, r_n\} \cup \{r_c\}$). The output place of $c_\downarrow$ is c (i.e., $c_\uparrow \bullet = \{c\}$).*
- *The initial marking M_0 of $\mathcal{Z}_c$ is emtpy (i.e., $\forall p \in P_c . M_0(p) = 0$) if the capability c is not offered in the initial state of the corresponding node. Otherwise, it contains exactly one token in r_c and in all places r_i denoting a requirement that is not assumed in the initial state of the corresponding node.*

Example 5.5. An example of controller (for a capability c connected to two requirements r_1 and r_2) is illustrated in Fig. 5.21. □

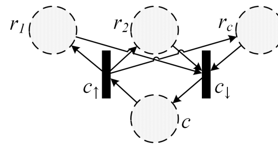


FIGURE 5.21: Example of *capability controller*.

We can now compose the nets modelling the management protocols of the nodes in the topology of a composite application by interconnecting them with the above introduced controllers. The composition is quite simple: We just collapse the open places corresponding to the same requirements and capabilities.

Definition 5.16 (Encoding of composite applications). *Let $A = \langle T, b \rangle$ be a composite application. We encode A with an open Petri net $\mathcal{Z}_A = \langle \mathcal{P}_A, I_A \rangle$, where $\mathcal{P}_A = \langle P_A, T_A, \bullet\cdot, \cdot\bullet, M_0 \rangle$, as follows.*

- *For each node $N \in T$, we encode its management protocol with an open Petri net $\mathcal{Z}_N$ obtained as shown in Def. 5.13.*

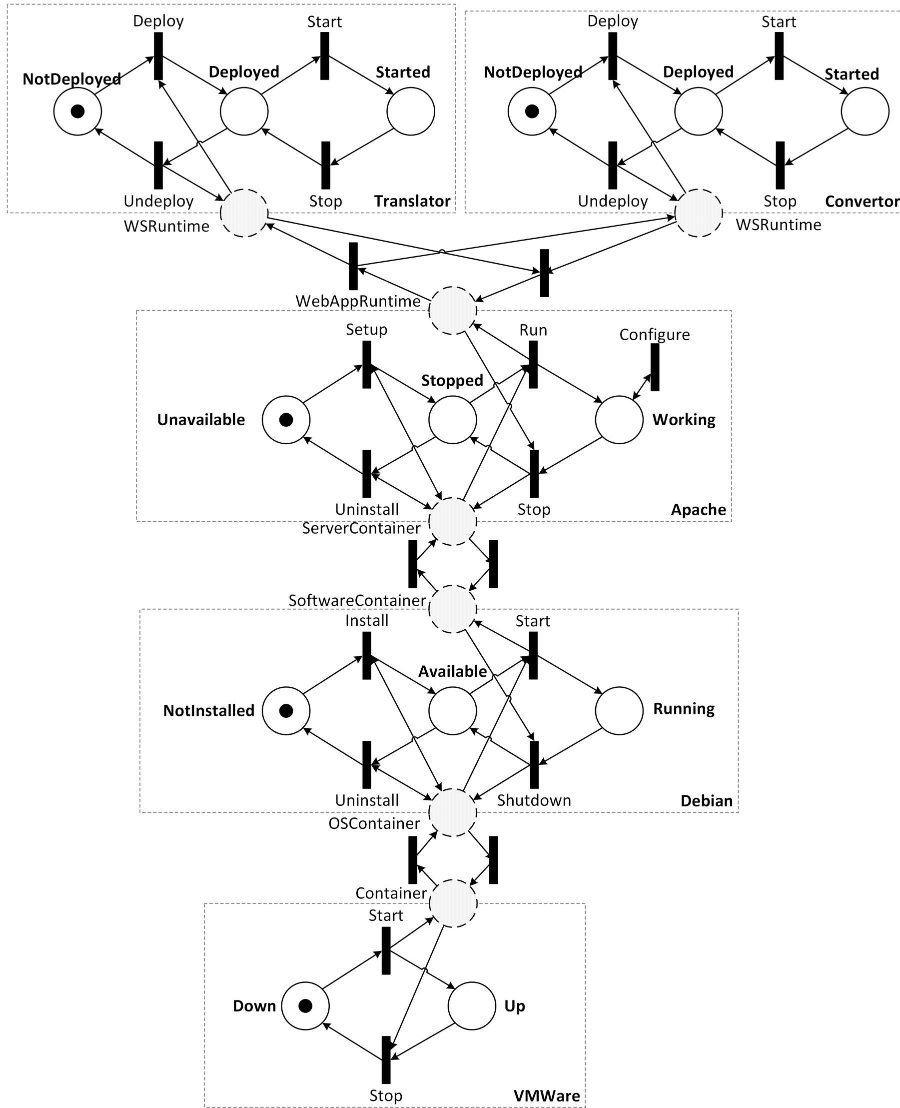


FIGURE 5.22: Petri net encoding for the motivating scenario in Sect. 5.1.

- *For each capability c exposed by a node $N \in T$, we create an open Petri net $\mathcal{Z}_c$ (acting as its controller) as shown in Def. 5.15.*
- *We then compose the above mentioned nets by taking their disjoint union and merging the places denoting the same requirement r or capability c.*
- *The initial marking M_0 is the union of the markings of the collapsed nets.*

Example 5.6. For example, Fig. 5.22 shows the net obtained for the motivating scenario described in Sect. 5.1. For the sake of readability, in the figure we omit, for each capability c, the place r_c of its controller. □

A very convenient property of the obtained encoding is that it is *safe* (i.e., the number of tokens in each place does not exceed one, for any marking M that is reachable from the initial marking M_0 [89]). To prove it, we need to further characterise the Petri net encoding we provided through Defs. 5.13, 5.15 and 5.16.

Property 5.1. *Let $A = \langle T, b \rangle$ be a composite application, and let $\mathcal{Z}_A$ be its Petri net encoding.*

$$\mathcal{Z}_A \text{ is safe.}$$

Proof. The property follows from the properties (i), (ii), and (iii) ensured by Lemma 5.1. More precisely, (i) proves that the internal places denoting node states can contain at most one token, (ii) proves that each open place denoting a capability c (as well as the corresponding place r_c) can contain at most one token, and (iii) proves that each open place denoting a requirement can contain at most one token. Therefore, all places in $\mathcal{Z}_A$ can contain at most one token (in any reachable marking), thus making the whole net safe [89]. □

Lemma 5.1. *Let $A = \langle T, b \rangle$ be a composite application, and let $\mathcal{Z}_A = \langle \mathcal{P}_A, I_A \rangle$ be the Petri net encoding of A, with $\mathcal{P}_A = \langle P_A, T_A, \bullet\cdot, \cdot\bullet, M_0 \rangle$. Let also M be a marking reachable from the initial marking M_0 of $\mathcal{Z}_A$. For each node $N_i = \langle S_{N_i}, R_{N_i}, C_{N_i}, O_{N_i}, \mathcal{M}_{N_i} \rangle$ (with $\mathcal{M}_{N_i} = \langle \overline{s}_{N_i}, \rho_{N_i}, \chi_{N_i}, \tau_{N_i} \rangle$) in T, the following properties hold:*

(i) $\exists s' \in S_{N_i}.M(s') = 1 \;\wedge\; \forall s \in S_{N_i}.s \neq s' \Rightarrow M(s) = 0$ *or, equivalently:*

$$\Sigma_{s \in S_{N_i}} M(s) = 1$$

(ii) Let s be the current state of a node N_i (i.e. $s \in S_{N_i} \wedge M(s) = 1$). For any capability $c \in C_{N_i}$, the number of tokens in the open places r_c and c is:

$$c \notin \chi(s) \Leftrightarrow M(c) + M(r_c) = 0$$
$$c \in \chi(s) \Leftrightarrow M(c) + M(r_c) = 1$$

(iii) Let s be the current state of a node N_i (i.e. $s \in S_{N_i} \wedge M(s) = 1$). For any requirement $r \in R_{N_i}$ bound to a capability c (i.e., $r \in$ binding(c, A)), the number of tokens in the open places r and r_c is:

$$r \notin \rho_{N_i}(s) \Leftrightarrow (M(r) = M(r_c) = 0) \vee (M(r) = M(r_c) = 1)$$
$$r \in \rho_{N_i}(s) \Leftrightarrow M(r) = 0 \wedge M(r_c) = 1$$

Proof. The proofs for (i), (ii), and (iii) are listed below.

(i) For each node N_i, the places denoting its states are internal to $\mathcal{Z}_A$. Hence, their input and output transitions are not changed by the merging process, which in turn means that only the net transitions (encoding the protocol transitions) of the same node N_i can add/remove tokens to/from them.

By construction, the above mentioned transitions always input exactly one token from an internal place and output exactly one token

to an internal place (potentially the same). This guarantees that the total number of tokens in the internal places of a single node cannot change:

$$\Sigma_{s \in S_{N_i}} M(s) = \Sigma_{s \in S_{N_i}} M'(s),$$

where M' is a marking reached by firing a transition in M.

The above, along with the fact that the initial marking M_0 of $\mathcal{Z}_A$ includes a token only in the places denoting the initial states of the nodes in T (i.e., for each node $N_i \in T$, $\Sigma_{s \in S_{N_i}} M_0(s) = 1$), implies that any sequence of firings starting from the initial marking will preserve exactly one token in the internal places denoting the states of each node.

(ii) In the initial marking M_0 of $\mathcal{Z}_A$ the property trivially descends from Defs. 5.13, 5.15, and 5.16.

Since the property holds for the initial marking, we can prove that it holds for every reachable marking, by showing that no transition can invalidate the property. We will thus consider it as invariant.

Consider the capability c of a node $N_i \in T$. The places mentioned in the property (i.e., c and r_c) are connected to the $c_\uparrow$ and $c_\downarrow$ transitions, and to the transitions of N_i that input/output a token to/from c. These are the only transitions that might affect the invariant, since the transitions connected to the requirements managed by the controller of c cannot change the marking of c nor that of r_c.

The $c_\uparrow$ and $c_\downarrow$ transitions cannot affect the invariant, since they do not change the total number of tokens in c and r_c. This is because, whenever $c_\uparrow$ fires, it removes one token from c, but it also adds one token to r_c (and to all of the other r_i places). Symmetrically, whenever $c_\downarrow$ fires, it removes one token from r_c (and from each of the other r_i places), but it also adds one token to c.

Thus, the only transitions that might invalidate the invariant are the transitions of the node N_i that input/output one token to/from c. Since all these transitions move a token from a state s to a state s', they can be classified as follows:

(a) c is either provided in both s and s' or in neither of them (i.e., $c \in \chi_{N_i}(s) \cap \chi_{N_i}(s') \vee c \notin \chi_{N_i}(s) \cup \chi_{N_i}(s')$);

(b) c is provided in s', but it is not provided in s (i.e., $c \in \chi_{N_i}(s') - \chi_{N_i}(s)$);

(c) c is provided in s, but it is not provided in s' (i.e., $c \in \chi_{N_i}(s) - \chi_{N_i}(s')$).

Each of these cases is consistent with the property that we want to prove.

(a) In the first case, transitions do not affect c at all, as (by construction) they are not even connected to c. They thus preserve the sum $M(c) + M(r_c)$, as well as the truth value of $c \in \chi_{N_i}(\cdot)$.

(b) In the second case, transitions lead to a state s' such that $c \in \chi_{N_i}(s')$, but they also add a token to c. If the invariant held before the transition (i.e., $M(c)+M(r_c) = 0$ with $M(s) = 1 \wedge c \notin \chi_{N_i}(s)$), it also holds after the transition, because the sum becomes $M(c)+M(r_c) = 1$ with $M(s') = 1 \wedge c \in \chi_{N_i}(s)$.

(c) The third case is precisely the opposite of the second one, since transitions lead to a state s' such that $c \notin \chi_{N_i}(s')$ and they remove a token from c. If the invariant held before the transition (i.e., $M(c) + M(r_c) = 1$ with $M(s) = 1 \wedge c \in \chi_{N_i}(s)$), then it also holds after the transition. The sum indeed becomes $M(c) + M(r_c) = 1$ with $M(s') = 1 \wedge c \notin \chi_{N_i}(s)$.

In conclusion, since the invariant holds for M_0 and none of the transitions can invalidate it, by induction (over the length of a firing sequence) it holds for any reachable marking.

(iii) The proof of the property follows the same line as the one for (ii). Namely, the property can be proved to hold for any reachable marking by induction over the length of a firing sequence, by showing that it holds for the initial marking M_0, and that none of the transitions can invalidate such property.

□

5.6.4 Analysing the management of a cloud application

The Petri net encoding of the management of a composite application A permits us defining what is a *valid plan* according to such management. Essentially, thanks to the encoding of capability controllers and to the way we compose these controllers with management protocol encodings, the obtained net ensures that no requirement can be assumed to hold if the corresponding capability is not provided, and that no capability can be removed if at least one of the corresponding requirements is assumed to hold. This permits to consider a sequence of operations as valid if and only if it corresponds to a firing sequence in the net encoding of A.

Definition 5.17 (Valid sequence of operations). *Let $A = \langle T, b \rangle$ be a composite application, and let $\mathcal{Z}_A = \langle \mathcal{P}_A, I_A \rangle$ be the Petri net encoding of A, with $\mathcal{P}_A = \langle P_A, T_A, \bullet\cdot, \cdot\bullet, M_0 \rangle$. A sequence $o_1 o_2 ... o_m$ of management operations is* valid *for A if and only if there is a firing sequence $t_1 t_2 \dots t_n$ (with $t_i \in T_A$) from the initial*

marking M_0 such that

$$o_1 \cdot o_2 \cdot \ldots \cdot o_m = \lambda(t_1) \cdot \lambda(t_2) \cdot \ldots \cdot \lambda(t_n),$$

where $\cdot$ indicates the concatenation operator[18] *and:*

$$\lambda(t) = \begin{cases} \epsilon & \text{if } t \text{ denotes a } c_\uparrow \text{ or } c_\downarrow \text{transition} \\ o & \text{if } t \text{ denotes a management protocol transition } \langle s, P, o, s' \rangle \end{cases}$$

We can easily extend the definition of validity from operation sequences to (workflow) plans, by constraining all their sequential traces to be valid.

Definition 5.18 (Valid plan). *Let $A = \langle T, b \rangle$ be a composite application, and let $\mathcal{Z}_A$ be its Petri net encoding. A plan P is* valid *for A if and only if all its sequential traces are valid sequences of operations for A (see Def.5.17).*

Example 5.7. Consider the deployment plans for the motivating scenario that we discussed in Sect. 5.1 (and displayed in Fig. 5.2). It is easy to see that plan (c) of is valid since all its sequential traces do have a corresponding firing sequence in the Petri net in Fig. 5.22. For instance,

VMWare:*Start Container*$_\uparrow$ *Debian*:*Install Debian*:*Start SoftwareContainer*$_\uparrow$ *Apache*:*Setup Apache*:*Run Apache*:*Configure WebAppRuntime*$_\uparrow$ *Convertor*:*Deploy Convertor*:*Start Translator*:*Deploy Translator*:*Start*

is a firing sequence corresponding to one of the sequential traces of plan (c).

Conversely, plans (a) and (b) in Fig. 5.2 are not valid as there are no corresponding firing sequences. Plan (a) is not valid since after firing

VMWare:*Start Container*$_\uparrow$ *Debian*:*Install Debian*:*Start SoftwareContainer*$_\uparrow$ *Apache*:*Setup*

transition *Apache*:*Configure* cannot be fired. It indeed requires a token in the place modelling the *Working* state of *Apache*, but that place is empty and it is not possible to add tokens to it without firing *Apache*:*Run*.

On the other hand, plan (b) is not valid since after firing

VMWare:*Start Container*$_\uparrow$ *Debian*:*Install*

transition *Apache*:*Setup* cannot fire. It requires a token in the place denoting the *ServerContainer* requirement, but that place is empty and it is not possible to add tokens to it without firing *SoftwareContainer*$_\uparrow$, which in turn cannot fire as it misses a token in the place denoting the *SoftwareContainer* capability of *Debian* (and no token can be added to such place without firing *Debian*:*Start*). □

However, the above Def. 5.18 does not ensure that all traces end up in the same setting of a composite application. Two different traces can reach two different markings with a different token assignment for the internal

[18] The empty string ϵ is the neutral element of $\cdot$, hence all transitions in controller nets are ignored (as $\lambda(t) = \epsilon$ when t denotes a $c_\uparrow$ or $c_\downarrow$ transition).

places. This would mean that, by differently inter-leaving the activities in a plan, the nodes in a composite application can end up in different states (thus potentially activating different capabilities and assuming different requirements). This is not acceptable in the management of composite applications [33], as we would expect a plan to have deterministic effects (independently of the inter-leaving of the activities that compose such plan). We thus define the notion of *deterministic* plans, after introducing that of internally equivalent markings.

Definition 5.19 (Equivalence of markings)**.** *Let* $\mathcal{Z} = \langle \mathcal{P}, I \rangle$*, with* $\mathcal{P} = \langle P, T, \bullet\cdot, \cdot\bullet, M_0 \rangle$ *be an open Petri net. Two markings* $M_1, M_2 : P \rightarrow \mathbb{N}$ *are* internally equivalent *(*$M_1 \equiv_M M_2$*) if and only if*

$$\forall p \in P \setminus I. M_1(p) = M_2(p)$$

Definition 5.20 (Deterministic plan)**.** *Let* $A = \langle T, b \rangle$ *be a composite application, and let* $\mathcal{Z}_A = \langle \mathcal{P}_A, I_A \rangle$ *be the Petri net encoding of* A*, with* $\mathcal{P}_A = \langle P_A, T_A, \bullet\cdot, \cdot\bullet, M_0 \rangle$*. Let also* P *be a valid plan for* A*.* P *is also* deterministic *if and only if for each pair* M_1, M_2 *of markings reached by executing two finite, complete*[19] *sequential traces of* P

$$M_1 \equiv_M M_2.$$

The effects of a plan on the states of the components forming a composite application, as well as on the requirements that are satisfied and the capabilities that are available, can then be directly determined from the marking that is reached performing the corresponding firing sequence. We thus first characterise the states, requirements, and capabilities that are active in a marking (Def. 5.21), and we then employ such characterization to list the effects of a deterministic plan (Remark 5.2).

Definition 5.21 (States, requirements, and capabilities active in a marking)**.** *Let* $A = \langle T, b \rangle$ *be a composite application, and let* $\mathcal{Z}_A = \langle \mathcal{P}_A, I_A \rangle$ *be the Petri net encoding of* A*, with* $\mathcal{P}_A = \langle P_A, T_A, \bullet\cdot, \cdot\bullet, M_0 \rangle$*. Let also* $N_i = \langle S_{N_i}, R_{N_i}, C_{N_i}, O_{N_i}, \mathcal{M}_{N_i} \rangle$*, with* $\mathcal{M}_{N_i} = \langle \overline{s}_{N_i}, \rho_{N_i}, \chi_{N_i}, \tau_{N_i} \rangle$*, be a node in* T*. Finally, let* M *be a marking.*

- *The* active states *in* M *are*

$$A_S^M = \{s \mid s \in P_A \setminus I_A \wedge M(s) = 1\}.$$

- *The* assumed requirements *in* M *are*

$$A_R^M = \{r \mid M(r) = 0 \wedge r \in b(c, A) \wedge M(r_c) = 1\}.$$

[19] A sequential trace for a plan P is *complete* if and only if its first and last operation correspond to an initial and to a final activity of P.

- *The* offered capabilities *in M are*

$$A_C^M = \{c \mid M(c) = 1 \vee M(r_c) = 1\}.$$

Remark 5.2. Let A be a composite application, and let $\mathcal{Z}_A$ be its Petri net encoding. Let also P be a deterministic *Plan*, and let M_0 and M be the initial marking and a marking equivalent to the markings reached by performing the (complete) sequential traces of P in M_0.

- The requirements that are assumed after P are A_R^M (where the newly assumed ones are $A_R^M \setminus A_R^{M_0}$), while those that are no more assumed are $A_R^{M_0} \setminus A_R^M$.
- The capabilities that are offered after P are A_C^M (where the newly added ones are $A_C^M \setminus A_C^{M_0}$), while those that are no more offered are $A_C^{M_0} \setminus A_C^M$.

□

Please note that it is possible to consider as initial marking any other (reachable) marking so as to analyse maintenance plans (starting from non-initial states) besides deployment plans. Obviously, the very same properties and techniques also apply in this case.

Additionally, various classical notions in the Petri net context assume a specific meaning in the context of managing composite applications. For example, the problem of finding whether there is a management plan which achieves a specific goal (e.g., bringing some components of an application to specific states or making some capabilities available) can be reduced in a straightforward way to the coverability problem [89] on the associated Petri net. To show it, we first define the notion of *goal*, that is a marking putting exactly one token in the places denoting the states and capabilities that have to be active.

Definition 5.22 (Goal). *Let $A = \langle T, b \rangle$ be a composite application, and let $N_i = \langle S_{N_i}, R_{N_i}, C_{N_i}, O_{N_i}, \mathcal{M}_{N_i} \rangle$ (with $\mathcal{M}_{N_i} = \langle \overline{s}_{N_i}, \rho_{N_i}, \chi_{N_i}, \tau_{N_i} \rangle$) be a node in T. A* goal *for planning in $\mathcal{Z}_A$ is a pair $\Gamma = \langle S_\Gamma, C_\Gamma \rangle$ such that*

(a) $S_\Gamma \subseteq \bigcup_i S_{N_i}$ is the set of states to be reached, and

(b) $C_\Gamma \subseteq \bigcup_i C_{N_i}$ is the set of capabilities to be offered.

A valid sequential plan P for A reaches the goal $\Gamma = \langle S_\Gamma, C_\Gamma \rangle$ if and only if

(a) $\forall s \in S_\Gamma . s \in S_{N_i} \Rightarrow s$ is the current state of N_i, and

(b) $\forall c \in C_\Gamma . c \in C_{N_i} \wedge s$ is the current state of $N_i \Rightarrow c \in \chi(s)$.

Proposition 5.1. *Let A be a composite application, and let $\mathcal{Z}_A$ be the Petri net encoding of A. Finding a valid sequential plan for A that reaches a goal Γ corresponds to solving a coverability problem in $\mathcal{Z}_A$.*

Proof. Let $\Gamma = \langle S_\Gamma, C_\Gamma \rangle$. We can easily build a marking $M_\Gamma : P_A \rightarrow \{0, 1\}$ as follows:

$$\forall p \in P_A . M_\Gamma(p) = \begin{cases} 1 & \text{if } p \in S_\Gamma \\ 1 & \text{if } p = r_c \wedge c \in C_\Gamma \\ 0 & \text{otherwise} \end{cases}$$

It follows that finding a sequential plan that reaches the goal Γ corresponds to solving the coverability problem for the marking M_Γ. □

Proposition 5.2. *Let A be a composite application, and let Γ be a goal. Finding a valid sequence of operations for A that reaches Γ can be solved with polynomial space.*

Proof. The proof directly follows from the following facts: (i) The Petri net encoding $\mathcal{Z}_A$ of A is safe, (ii) finding a sequential plan in $\mathcal{Z}_A$ that reaches Γ corresponds to solving a coverability problem, and (iii) coverability in safe Petri nets is PSPACE-complete [41]. □

Another classical notion in the Petri net context that assumes a specific meaning is that of *reversibility* [89]: The Petri net encoding of a composite application A is *reversible* if and only if it is always possible to softly reset the application, i.e. if whatever (valid) sequence of operations we perform, we can always get back to the initial state of A by performing another (valid) sequence of operations (see Def. 5.10).

Proposition 5.3. *Let A be a composite application, and let $\mathcal{Z}_A$ be the Petri net encoding of A.*

A is softly resettable $\Leftrightarrow$ $\mathcal{Z}_A$ is reversible.

Proof. The condition given by Def. 5.10 can be rewritten as follows: A is softly resettable if and only if for each valid sequence $o_1 o_2 ... o_n$, we can always determine a longer valid sequence $o_1 o_2 ... o_m o_{m+1} ... o_{m+n}$ such that by firing it in the initial global state $\overline{G}$ we end up in the same configuration $\overline{G}$.

Notice that $\overline{G}$ corresponds to the initial marking of the Petri net encoding $\mathcal{Z}_A$, and that a valid sequence of operations corresponds to a firing sequence in $\mathcal{Z}_A$. Thus, the condition for soft resettability corresponds to saying that whatever firing sequence we can perform in the initial marking, we can always find a longer firing sequence that (starts and) ends up in the initial marking. This in turn corresponds to saying that $\mathcal{Z}_A$ is reversible (since whatever marking we can reach with a sequence of firings, we can always come back to the initial marking). □

5.6.5 Remarks

In this section we have presented an alternative semantics for management protocols (Def. 5.1). It is easy to prove that, for a composite application A, the set of plans that are considered valid for A by both the semantics we presented here and that presented in Sect. 5.3 is the same.

Proposition 5.4 (Equivalence of semantics)**.** *Let $A = \langle T, b \rangle$ be a composite application, and let $\overline{G}$ be its initial global state. The valid plans for A in $\overline{G}$ determined according to Def. 5.8 are the same as those determined according to Def. 5.18.*

Proof. Both Def. 5.8 and Def. 5.18 say that a plan is valid if and only if all its sequential traces are valid (by referring to Def. 5.7 and to Def. 5.17, respectively). Hence, to prove that the sets of valid plans determined according to Def. 5.8 and according to Def. 5.18 are equal, we just need to prove the following lemma. □

Lemma 5.2. *Let $A = \langle T, b \rangle$ be a composite application, and let $\overline{G}$ be its initial global state. The valid sequences of operations for S in $\overline{G}$ determined according to Def. 5.7 are the same as those determined according to Def. 5.17.*

Proof (sketch). The thesis can be proved by inductively constructing the sets V_1 and V_2 (containing the valid sequences of operations determined according to Def. 5.7 and to Def. 5.17, respectively), and by showing that $V_1 = V_2$.
$[V_1^0 = V_2^0]$ Let $\mathcal{Z}_A$ be the Petri net encoding of A. One can readily check that, by construction of $\mathcal{Z}_A$, (a) the initial marking M_0 of $\mathcal{Z}_A$ denotes the initial global state $\overline{G}$ of A, and (b) if we can fire a transition in $\mathcal{Z}_A$, this means that by performing the corresponding operation we preserve the consistency of the global state, and vice versa[20]. From (a) and (b), it follows that the set V_1^0 and V_2^0 containing the operations that can be fired in the initial global state of A (according to Def. 5.7 and to Def. 5.17, respectively) are equal.
$[V_1^n = V_2^n \Rightarrow V_1^{n+1} = V_2^{n+1}]$ Suppose now that the sets V_1^n and V_2^n (containing the valid sequences of — at most n — operations determined according to Def. 5.7 and according to Def. 5.17) are equal, i.e. $V_1^n = V_2^n$. Consider a sequence of operations $o_1..o_m \in V_1^n$ (with $m \leq n$), and suppose that M_m and G_m are the marking of $\mathcal{Z}_A$ and the global state of A reached by performing the sequence of operations in $o_1..o_m$. One can readily check that the M_m denotes the G_m, since the components in A will reach the same internal states by performing the same sequence of operations $o_1..o_m$ (independently of the employed semantics). This means that, if the transition denoting an operation o_{m+1} can be fired in M_m (maybe after performing

[20]This follows from the facts that the transitions in $\mathcal{Z}_A$ are built in such a way that a capability cannot be removed if a corresponding requirement is assumed, and that the global state becomes inconsistent after firing if transition that removes a capability connected to an assumed requirement.

some transitions in the capability controllers), then o_{m+1} can be performed also in G_m, and vice versa. Hence:

$$o_1..o_m o_{m+1} \text{ can be added to } V_1^{n+1} \Leftrightarrow o_1..o_m o_{m+1} \text{ can be added to } V_2^{n+1}.$$

As the above property obviously holds for any sequence of operation in V_1^n and for each operation available in A, it follows that the sets containing the valid sequences of at most $n+1$ operations determined according to both considered semantics are equal, i.e. $V_1^{n+1} = V_2^{n+1}$. □

We have also discussed how, by encoding management protocols into open Petri nets, we can exploit existing results to accomplish some of the analyses we want to carry out. For instance, the problem of finding whether there is a plan which achieves a specific goal can be reduced to the coverability problem, or the problem of determining whether a composite application is softly resettable can be solved by checking whether its encoding in open Petri nets is reversible.

On the other hand, the proposed Petri net encoding is built in such a way that it does not permit performing an operation if it can cause a violation of the consistency among requirements and capabilities. For instance, it is not possible to simulate the execution of an operation that removes a capability from a node while at least another node is relying on such capability, simply because the place denoting the capability to be removed is an input place for the corresponding transition and it is missing a token. As a consequence, it is not possible to determine the effects of removing such capability. The same does not hold for the labelled transition system-based semantics we presented in Sect. 5.3.

The above, along with the fact that having a "direct semantics" would avoid to recompute the Petri net encoding of an application whenever a node is added or removed from such application, makes the semantics we presented in Sect. 5.3 a better candidate for being extended to model and analyse faults (as we show in Chapter 7). This is why, in the following, we will focus on the labelled transition system-based semantics.

Chapter 6

Behaviour-aware matching of cloud applications

The *exact* and *plug-in* matching presented in Chapter 3 permit reusing available service templates to concretely implement desired node types (by checking that all features of the latter are provided by the former). Such techniques however do not take into account the behaviour of management operations, i.e. they do not check whether the behaviour of an available service template is compatible with the behaviour of the desired node type.

We hereby exploit the behaviour information in management protocols to extend such notions[1]. More precisely, after introducing a shorthand notation for management protocols (see Sect. 6.1), we define when a desired management protocol can be "simulated" [107] by another available protocol, and we exploit such notion of simulation to extend the conditions constraining exact and plug-in matching (see Sect. 6.2).

We then relax the notion of simulation into that of f-simulation, where f is a function associating each transition in the desired management protocol with a sequence of transitions in the available protocol. This permits further relaxing the notion of plug-in matching, by allowing to match the management operations of a desired node type with sequences of operations of an available service template. We also describe flexibly plug-in matched service templates can be suitably adapted so as to be employed in place of desired node types (see Sect. 6.3).

Finally, we introduce a coinductive [71] procedure that permits computing the function f determining an f-simulation among two management protocols. We also assess such procedure, by proving that it is sound and complete (see Sect. 6.4).

[1] Analogous extensions can be defined for *renaming-based* and *white-box* matching. Due to reasons of readability, and since their extensions are very similar to that of plug-in matching, we hereby focus on extending exact and plug-in matching.

6.1 A shortand notation for management protocols

To simplify notation throughout this chapter, we define an *intensional* operational semantics of the management protocol of a single component (viz., a TOSCA node type). The intensional semantics models all possible sequences of management operations that could be performed on a component if the conditions on the needed requirements are satisfied by the environment. Formally, the intensional semantics of the management protocol of a node type N can be defined by a labelled transition system over configurations that are the states of N.

Definition 6.1 (Intensional semantics of management protocols)**.** *Let $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$ be a node type, with $\mathcal{M}_N = \langle \overline{s}_N, \rho_N, \chi_N, \tau_N \rangle$. The* intensional semantics *of the management protocol $\mathcal{M}_N$ of N is modelled by a labelled transition system whose set of configurations is S_N and where the transition relation is defined by the following inference rule:*

$$\frac{N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle \qquad \mathcal{M}_N = \langle \overline{s}_N, \rho_N, \chi_N, \tau_N \rangle \quad \langle s, P, X, o, s' \rangle \in \tau_N}{s \xrightarrow{\langle P,X,o \rangle}_N s'}$$

Intuitively, a transition $s \xrightarrow{\langle P,X,o \rangle}_N s'$ denotes that operation o can be executed on N when N is in state s, and under the hypothesis that the requirements in P are satisfied, making N evolve into state s'. During the transition, it is guaranteed that N continues to offer the capabilities in X.

6.2 Simulation-based matching

Consider a node type $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$, where S_N, R_N, C_N, and O_N are respectively the sets of its states, requirements, capabilities, and management operations, and where $\mathcal{M}_N = \langle \overline{s}_N, \rho_N, \chi_N, \tau_N \rangle$ is the management protocol of N. Consider also a service template $S = \langle S_S, R_S, C_S, O_S, \mathcal{M}_S \rangle$, with $\mathcal{M}_S = \langle \overline{s}_S, \rho_S, \chi_S, \tau_S \rangle$. In order to extend the notions of *exact* and *plug-in* matching (Defs. 3.1 and 3.6), we formally define when the management protocol $\mathcal{M}_S$ of S can simulate [107] (the behaviour defined by) the management protocol $\mathcal{M}_N$ of N.

6.2.1 Simulation of management protocols

Intuitively speaking, $\mathcal{M}_N$ is simulated by $\mathcal{M}_S$ if and only if the initial state of $\mathcal{M}_N$ is simulated by the initial state of $\mathcal{M}_S$. A state $s_N \in S_N$ is simulated by a state $s_S \in S_S$ if and only if (a) the requirements needed by s_N include all those needed by s_S, (b) the capabilities offered by s_N are included in those offered by s_S, and (c) for each transition from s_N to s'_N, there is a

"compatible" transition starting from s_S (i.e., a transition performing the same operation o, not needing additional requirements, providing at least the same capabilities, and leading to a state s'_S that simulates s'_N).

Definition 6.2 (Simulation of management protocols). *Let $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$ be a node type, with $\mathcal{M}_N = \langle \overline{s}_N, \rho_N, \chi_N, \tau_N \rangle$. Let also $S = \langle S_S, R_S, C_S, O_S, \mathcal{M}_S \rangle$ be a service template, with $\mathcal{M}_S = \langle \overline{s}_S, \rho_S, \chi_S, \tau_S \rangle$.*

A state $s_N \in S_N$ is simulated *by $s_S \in S_S$ ($s_N \sqsubseteq s_S$) iff*

(a) $\rho_N(s_N) \supseteq \rho_S(s_S)$,

(b) $\chi_N(s_N) \subseteq \chi_S(s_S)$, *and*

(c) $s_N \xrightarrow{\langle P_N, X_N, o \rangle}_N s'_N$
implies
$\exists s_S \xrightarrow{\langle P_S, X_S, o \rangle}_S s'_S : P_N \supseteq P_S \wedge X_N \subseteq X_S \wedge s'_N \sqsubseteq s'_S$.

A management protocol $\mathcal{M}_N$ is simulated *by another management protocol $\mathcal{M}_S$ (viz., $\mathcal{M}_N \sqsubseteq \mathcal{M}_S$) iff $\overline{s}_N \sqsubseteq \overline{s}_S$.*

6.2.2 Behaviour-aware exact and plug-in matching

The notion of simulation permits extending those of *exact* and *plug-in* matching introduced in Chapter 3.

To check whether a service template S exactly matches a node type N, we now check whether S syntactically exactly matches N (see Def. 3.1), and whether the management protocol of S simulates that of N.

Definition 6.3 (Behaviour-aware exact matching). *Let $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$ be a node type, and let $S = \langle S_S, R_S, C_S, O_S, \mathcal{M}_S \rangle$ be a service template. S* exactly *matches N ($S \equiv_b N$) iff*

$$S \equiv N \ \wedge \ \mathcal{M}_N \sqsubseteq \mathcal{M}_S.$$

Analogously, to check whether a service template S plug-in matches a node type N, we check whether S syntactically plug-in matches N (see Def. 3.6), and whether the management protocol of S simulates that of N.

Definition 6.4 (Behaviour-aware plug-in matching). *Let $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$ be a node type, and let $S = \langle S_S, R_S, C_S, O_S, \mathcal{M}_S \rangle$ be a service template. S* plug-in *matches N ($S \simeq_b N$) iff*

$$S \simeq N \ \wedge \ \mathcal{M}_N \sqsubseteq \mathcal{M}_S.$$

Example 6.1. Consider the *Server* node type and the *Tomcat* service template in Fig. 6.1. *Tomcat* syntactically plug-in matches *Server*[2] (viz., *Server* $\simeq$ *Tomcat*).

[2]For simplicity, please assume that same-named requirements, capabilities, and operations satisfy the syntactical plug-in matching conditions given in Chapter 3.

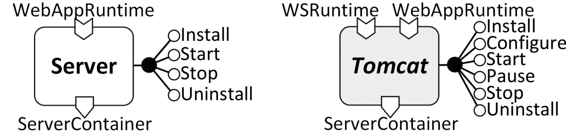


FIGURE 6.1: Examples of node type and service template.

Tomcat can thus be adapted to exactly match *Server* (see Sect. 3.1.2). We build a new service template having *Tomcat* as its only node, and exposing (via its boundary definitions) the same requirement, capability, and operations as the target *Server* node type. The resulting service template is shown in Fig. 6.2.

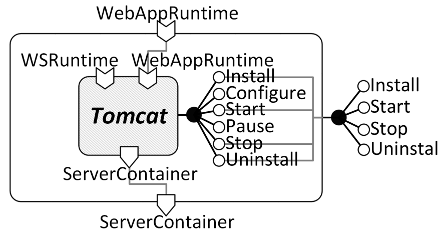


FIGURE 6.2: Example of adaptation of a syntactically plug-in matched service template.

Consider now the management protocols in Fig. 6.3, where $\mathcal{M}_{\text{Ser}}$ is the management protocol for *Server*, while $\mathcal{M}_{\text{Tom}}$ and $\mathcal{M}'_{\text{Tom}}$ are two (alternative) management protocols for *Tomcat*.

It is easy to see that $\mathcal{M}_{\text{Ser}} \sqsubseteq \mathcal{M}_{\text{Tom}}$ since *Unavailable* $\sqsubseteq$ *NotInstalled*. It follows that (with $\mathcal{M}_{\text{Tom}}$) *Tomcat* plug-in matches *Server* (viz., *Tomcat* $\simeq_b$ *Server*), and that *Tomcat* can still be adapted as shown in Fig. 6.2 to exactly match *Server*. The adaptation now also ensures that the management operations of the adapted service template have the same behaviour as those of the desired node type.

The same does not hold if the management protocol of *Tomcat* is $\mathcal{M}'_{\text{Tom}}$. For instance, by performing *Install* in their initial states, $\mathcal{M}'_{\text{Tom}}$ and $\mathcal{M}_{\text{Ser}}$ respectively reach the states *Installed* and *Stopped*, and *Installed* $\not\sqsubseteq$ *Stopped*. This is because there is no transition starting from *Installed* that corresponds to the operation *Start* of *Server*. Instead, by making the *Install* operation of *Server* correspond to the sequencing of the operations *Install* and *Configure* of *Tomcat*, the aforementioned problem would not be raised, since in *Configured* state it is possible to fire *Start*. This means that a less strict definition of (operation) matching should allow *Tomcat* to match *Server* (with $\mathcal{M}'_{\text{Tom}}$ as management protocol). □

6.3 Flexible simulation-based matching

We now further extend the definition of matching in order to identify larger sets of service templates that can be adapted to exactly match a desired node type $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$. More precisely, we relax the definition of plug-in matching so that, given a service template $S = \langle S_S, R_S, C_S, O_S, \mathcal{M}_S \rangle$, we permit matching (and substituting) the operations in O_N with

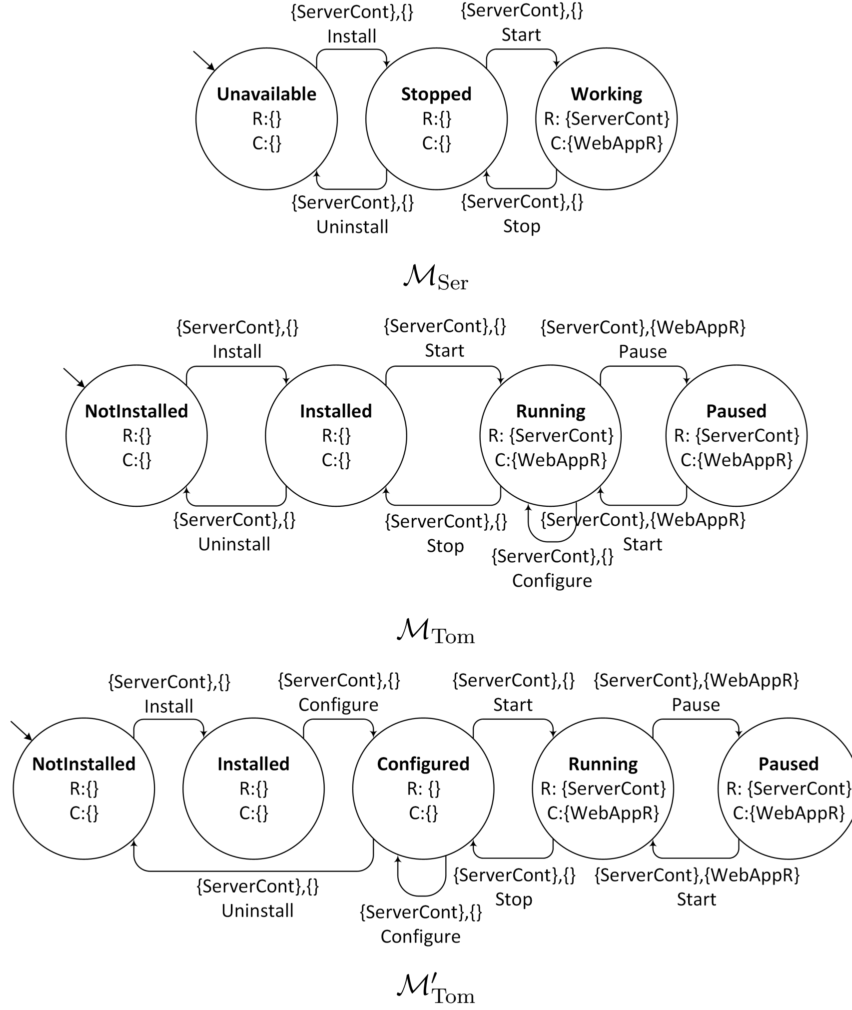


FIGURE 6.3: Example of management protocols.

sequences of operations in O_S, based upon their effects on states, requirements and capabilities.

6.3.1 Flexible-simulation of management protocols

We first relax the notion of management protocol simulation (Def. 6.2) by allowing to simulate each transition of the target management protocol $\mathcal{M}_N$ with sequences of transitions of the available management protocol $\mathcal{M}_S$. To do so, we first extend the intensional semantics of $\mathcal{M}_S$, by adding the transitions which permit remaining in the same state by performing an empty sequence ϵ of operations (without changing the conditions on requirements and capabilities), and which permit moving from a state to another by performing non-empty sequences of operations. While for singleton sequences the rule is trivial, for sequences of at least two operations we need the following rule: If w_1 permits transiting from state s to state s'' by assuming the requirements in H_1 and by maintaining the capabilities in G_1, and if w_2 permits transiting from state s'' to state s' by assuming the requirements in

H_2 and by maintaining the capabilities in G_2, then w_1w_2 permits transiting from s to s' by assuming $H_1 \cup H_2$ and by maintaining $G_1 \cap G_2$.

Definition 6.5 (n-step intensional semantics). *Let $S = \langle S_S, R_S, C_S, O_S, \mathcal{M}_S \rangle$ be a service template. The* n-step intensional semantics *of the management protocol $\mathcal{M}_S$ of S is modelled by a labelled transition system whose set of configurations is S_S and whose transition relation is defined by the following inference rules:*

$$\frac{-}{s \xRightarrow{\langle \rho_S(s), \chi_S(s), \epsilon \rangle}_S s} \qquad \frac{s \xrightarrow{\langle P, X, o \rangle}_S s'}{s \xRightarrow{\langle P, X, o \rangle}_S s'}$$

$$\frac{s \xRightarrow{\langle P_1, X_1, w_1 \rangle}_S s'' \ \wedge \ s'' \xRightarrow{\langle P_2, X_2, w_2 \rangle}_S s'}{s \xRightarrow{\langle P_1 \cup P_2, X_1 \cap X_2, w_1 w_2 \rangle}_S s'}$$

Remark 6.1. The transition system $\Rightarrow_S$ from Def. 6.5 actually defines the reflexive and transitive closure of the transition system $\rightarrow_S$. One can readily check that whenever $\rightarrow_S$ is well-formed also $\Rightarrow_S$ is well-formed. This ensures that all the intermediates states that are reached during the execution of a transition

$$s \xRightarrow{\langle P, X, w \rangle}_S s'$$

offer at least capabilities X and require at most requirements P. This meets the intuition behind the label X introduced in Sect. 5.2: X are all the capabilities that are maintained available *during* a transition. □

According to Def. 6.5, the transition system $\Rightarrow_S$ generates infinite branching whenever the corresponding protocol features some loops. In order to obtain a finitary description of it, we restrict to consider only *minimal* sequences of operations.

Definition 6.6 (Minimal n-step intensional semantics). *Let $S = \langle S_S, R_S, C_S, O_S, \mathcal{M}_S \rangle$ be a service template. The* minimal n-step intensional semantics *of the management protocol $\mathcal{M}_S$ of S is modelled by a labelled transition system whose set of configurations is S_S and whose transition relation is defined by the following inference rule:*

$$\frac{s \xRightarrow{\langle P, X, w \rangle}_S s' \ \wedge \ \nexists P_1 \subseteq P, X_1 \supseteq X, w_1 w_2 = w : s \xRightarrow{\langle P_1, X_1, w_1 \rangle}_S s'}{s \bullet\!\!\xRightarrow{\langle P, X, w \rangle}_S s'}$$

We now relax the notion of simulation (Def. 6.2) into that of f-simulation, where $f : S_N \times S_S \times O_N \rightarrow O_S^*$ is a function associating each transition in the target management protocol $\mathcal{M}_N$ with a (possibly empty) sequence of transitions in the available management protocol $\mathcal{M}_S$.

Intuitively speaking, $\mathcal{M}_N$ can be f-simulated by $\mathcal{M}_S$ if and only if the initial state of $\mathcal{M}_N$ can be f-simulated by the initial state of $\mathcal{M}_S$. A state $s_N \in S_N$ is in turn f-simulated by a state $s_S \in S_S$ if and only if (a) the

requirements needed by s_N contain all those needed by s_S, (b) the capabilities offered by s_N are contained in those offered by s_S, and (c) for each transition starting from s_N, there is a transition in $\bullet\!\!\Rightarrow_S$ starting from s_S, not needing additional requirements, providing at least the same capabilities, and leading to a state s'_S that in turn f-simulates s'_N.

Definition 6.7 (f-simulation of management protocols). *Let $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N\rangle$ be a node type, with $\mathcal{M}_N = \langle \overline{s}_N, \rho_N, \chi_N, \tau_N\rangle$. Let also $S = \langle S_S, R_S, C_S, O_S, \mathcal{M}_S\rangle$ be a service template, with $\mathcal{M}_S = \langle \overline{s}_S, \rho_S, \chi_S, \tau_S\rangle$.*

*A state $s_N \in S_N$ is f-*simulated *by a state $s_S \in S_S$ ($s_N \sqsubseteq_f s_S$) iff the following conditions hold.*

(a) $\rho_N(s_N) \supseteq \rho_S(s_S)$*,*

(b) $\chi_N(s_N) \subseteq \chi_S(s_S)$*, and*

(c) $s_N \xrightarrow{\langle P_N, X_N, o\rangle}_N s'_N$
implies
$\exists s_S \overset{\langle P_S, X_S, f(s_N, s_S, o)\rangle}{\bullet\!\!\Longrightarrow}_S s'_S : P_N \supseteq P_S \wedge X_N \subseteq X_S \wedge s'_N \sqsubseteq_f s'_S$.

*A management protocol $\mathcal{M}_N$ is f-*simulated *by a management protocol $\mathcal{M}_S$ ($\mathcal{M}_N \sqsubseteq_f \mathcal{M}_S$) iff $\overline{s}_N \sqsubseteq_f \overline{s}_S$.*

Remark 6.2. A similar notion could be defined by replacing $\bullet\!\!\Rightarrow_S$ with $\Rightarrow_S$ (Def. 6.5). This would put weaker constraints on f, as the transition system $\Rightarrow_S$ is much larger than $\bullet\!\!\Rightarrow_S$. Nonetheless any f satisfying the weaker simulation constraints would have an associated f' fulfilling the stricter ones.

We have selected $\bullet\!\!\Rightarrow_S$ to have a decidable f-simulation, since $\bullet\!\!\Rightarrow_S$ can be proved to be finite and computable. □

6.3.2 Flexible plug-in matching

It is easy to see that the notion of f-simulation (Def. 6.7) supports a more *flexible* form of matching than simulation (Def. 6.2), by permitting to match an operation with (different) sequences of operations.

Definition 6.8 (Flexible plug-in matching). *Let $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N\rangle$ be a node type, and let $S = \langle S_S, R_S, C_S, O_S, \mathcal{M}_S\rangle$ be a service template. S* flexibly plug-in *matches N ($S \lesssim_b N$) iff conditions 1—4 in Def. 3.6 hold, and*

$$\mathcal{M}_N \sqsubseteq_f \mathcal{M}_S.$$

Remark 6.3. If an available service template S *plug-in* matches a node type N, then S also *flexibly plug-in* matches N, i.e.

$$S \simeq_b N \Rightarrow S \lesssim_b N$$

This is because both plug-in and flexible plug-in constrain the same conditions on requirements and capabilities, and because management protocol simulation corresponds to f-simulation when f is the identity function. □

It is worth noting that, while the adaptation technique for (syntactically) plug-in matched service templates presented in Chapter 3 can be directly applied to requirements and capabilities, management operations need now to be adapted by taking into account sequences. We hence map the operations exposed on the boundaries of the adapted service template onto plans composing the operation of the flexibly plug-in matched service template. More precisely, since each operation to be matched can be associated with a different sequence according to f (and depending on the states of the target node type and matched service template), each operation exposed by the adapted service template is now associated with a conditional workflow encoding all the mappings given by f.

Example 6.2. Consider again the *Server NodeType* and the *Tomcat ServiceTemplate* in Fig. 6.1. It is now easy to see that *Tomcat* flexibly plug-in matches *Server* (i.e., *Server* $\lesssim_b$ *Tomcat*), even if we consider the management protocols $\mathcal{M}_{\text{Ser}}$ and $\mathcal{M}'_{\text{Tom}}$ in Fig. 6.3. This is because $\mathcal{M}_{\text{Ser}}$ is f-simulated by $\mathcal{M}_{\text{Tom}}$, where f is the identity function, except that for mapping the *Install* operation of *Server* with the sequencing of the operations *Install* and *Configure* of *Tomcat*.

It follows that we can adapt *Tomcat* as shown in Fig. 6.4. Namely, we cre-

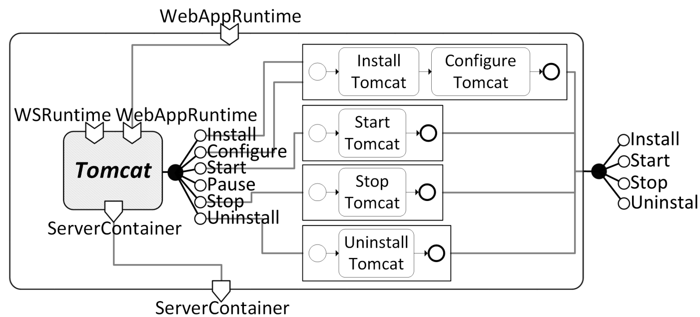


FIGURE 6.4: Example of adaptation of a flexibly plug-in matched service template.

ate a new service template containing *Tomcat* as its only node, and exposing on its boundary definitions the features of the *Server* node type to be matched. We then map requirements and capabilities as shown in Sect. 3.1.2. Finally, we implement each operation exposed by the adapted service template with workflow plans built according to the mappings given by f. Namely, the *Install* operation is implemented by a sequential plan which invokes the *Tomcat*'s operations *Install* and *Configure*. Each other operation is implemented by plans containing a single invocation to the homonym *Tomcat*'s operation (e.g., *Start* is implemented by a plan which only invokes the *Start* operation of *Tomcat*). □

It is worth noting that Example 6.2 shows a static translation: Each operation exposed on the boundaries of the adapted service template is implemented by a plan that is independent from the current states of the target

node type and of the matched service template. According to the definition of f-simulation (Def. 6.7) this is not always the case, since f may depend on the current states of the target node type and of the available service template. In the next section we present an algorithm which computes *all* the possible f, from which it is trivial to extract a static translation (if it exists). If no such translation exists, the adapter would need some additional logic, namely it should include some conditional statements to track the state of the target node type and/or of the reused service template.

6.4 Computing a flexible simulation

Consider a service template $S = \langle S_S, R_S, C_S, O_S, \mathcal{M}_S \rangle$ and a node type $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_S \rangle$, where $\mathcal{M}_S = \langle \overline{s}_S, \rho_S, \chi_S, \tau_S \rangle$ and $\mathcal{M}_N = \langle \overline{s}_N, \rho_N, \chi_N, \tau_N \rangle$ are the corresponding management protocols. According to Def. 6.8, to check whether S flexibly plug-in matches N we need to check whether $\mathcal{M}_S$ f-simulates $\mathcal{M}_N$ (i.e., $\mathcal{M}_S \sqsubseteq_f \mathcal{M}_N$). We hereby illustrate a coinductive [71] procedure that permits computing all functions f such that $\mathcal{M}_S$ f-simulates $\mathcal{M}_N$ (if any).

6.4.1 A coinductive approach to compute f-simulations

Let us denote with $W_S \subseteq O_S^*$ set of all minimal operation sequences in $\mathcal{M}_S$ (Def. 6.6). We now sketch a coinductive algorithm capable of finding all functions

$$f : S_N \times S_S \times O_N \to W_S$$

such that $\mathcal{M}_S \sqsubseteq_f \mathcal{M}_N$. Intuitively speaking, the algorithm starts by permitting to map each operation in O_N with any sequence of operations in W_S, and iteratively refines the mapping by removing the mappings leading to states that do not f-simulate (for any f). This process continues until the mapping cannot be refined any more.

More precisely, the algorithm employs a $l \times m \times n$ matrix F, where l is the number of states in S_N (i.e., $l = |S_N|$), m is the number of states in S_S (i.e., $m = |S_S|$), and n is the number of operations in O_N (i.e., $n = |O_N|$). Each entry $F[i, j, o]$ stores the set of all words $w \in W_S$ such that, for some f, mapping (i, j, o) into w would potentially allow j to f-simulate i (i.e., $i \sqsubseteq_f j$). Intuitively, the matrix F represents an element in the lattice of all functions $S_N \times S_S \times O_N \to 2^{W_S}$, and the algorithm consists of a greatest fixpoint computation on such lattice (i.e., F starts from the top of the lattice, and each refinement makes it become a lower element — until it cannot be lowered any more).

Initially, there is no information about f-simulation, and the algorithm can only check whether two states $i \in S_N$ and $j \in S_S$ are compatible

in terms of requirements and capabilities (i.e., whether $\rho_N(i) \supseteq \rho_S(j) \wedge \chi_N(i) \subseteq \chi_S(j)$). If this is the case, then j is a candidate to simulate i, and the algorithm maps each operation $o \in O_N$ with any operation sequence in W_S (i.e., $F_0[i,j,o] = W_S$). Otherwise, there is no way to simulate i with j, and thus there is no way to map the operations in O_N onto sequences in W_S.

$$F_0[i,j,o] = \begin{cases} W_S & \text{if } \rho_N(i) \supseteq \rho_S(j) \wedge \chi_N(i) \subseteq \chi_S(j) \\ \varnothing & \text{otherwise} \end{cases}$$

At each step $k+1$, F_k is refined by removing all the mappings that lead to states that do not f-simulate. More precisely, given $i \in S_N$, $j \in S_S$, and $o \in O_N$, $F_{k+1}[i,j,o]$ is obtained by restricting $F_k[i,j,o]$ to those w such that if i can go in i' with o, then j can go in j' with w, and j' is a candidate to simulate i' (i.e., $\forall o' \in O_N . F_k[i',j',o'] \neq \varnothing$).

$$F_{k+1}[i,j,o] = \{w \in F_k[i,j,o] \mid \forall i \xrightarrow{\langle P_N, X_N, o\rangle}_N i' . \exists j \overset{\langle P_S, X_S, w\rangle}{\Longrightarrow}_S j' : \\ P_N \supseteq P_S \wedge X_N \subseteq X_S \wedge \forall o' \in O_N . F_k[i',j',o'] \neq \varnothing\}$$

This iterative process stops when the matrix F cannot be refined any more, i.e. when $F_{k+1} = F_k$. By definition, when $F_{k+1} = F_k$, we reached the maximum fixpoint F.

- If there is at least one operation in O_N that cannot be mapped in the starting states (i.e., $\exists o \in O_N . F[\overline{s}_N, \overline{s}_S, o] = \varnothing$), then there is no function f such that the starting states f-simulate (i.e., $\overline{s}_S \not\sqsubseteq_f \overline{s}_N$). This in turn implies that $\mathcal{M}_S$ does not f-simulate $\mathcal{M}_N$ (i.e., $\mathcal{M}_S \not\sqsubseteq_f \mathcal{M}_N$).
- Otherwise, we can extract one of the functions f (such that $\mathcal{M}_S \sqsubseteq_f \mathcal{M}_N$) by simply selecting one of the possible mappings for each operation $o \in O_N$ and for each pair of states $(i,j) \in S_N \times S_S$.

6.4.2 Properties of the approach

We hereby prove (by means of coinduction [71]) that the algorithm computing a f-simulation (see Sect. 6.4.1) is terminating, sound and complete.

Termination

The termination of the algorithm to compute a f-simulation follows trivially from its rules for initialising and refining the matrix F.

Proposition 6.1. *The algorithm presented in Sect. 6.4.1 always* terminates*.*

Proof. The algorithm consists in an iterative refinement process, which stops whenever the matrix F cannot be refined any more, i.e. $F_{k+1} = F_k$. This is guaranteed to happen, because of the following facts:

- All entries of the matrix F_0 are initialised with finite sets (viz., either they contain the set W_S of all minimal operation sequences, or they contain the empty set).
- At every step, each entry $F_k[i, j, o]$ either shrinks or stays the same, and $F_k[i, j, o]$ is lower-bounded by $\varnothing$.

□

Soundness and completeness

Consider a service template $S = \langle S_S, R_S, C_S, O_S, \mathcal{M}_S \rangle$ and a node type $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$, whose management protocols are $\mathcal{M}_S = \langle \overline{s}_S, \rho_S, \chi_S, \tau_S \rangle$ and $\mathcal{M}_N = \langle \overline{s}_N, \rho_N, \chi_N, \tau_N \rangle$, respectively. f-simulation is defined (in Def. 6.7) as a relation such that, for all pairs $(s_N, s_S) \in S_N \times S_S$, satisfies the following constraint:

$$
\begin{aligned}
s_N \sqsubseteq_f s_S :=& \rho_N(s_N) \supseteq \rho_S(s_S) \wedge \\
& \chi_N(s_N) \subseteq \chi_S(s_S) \wedge \\
& (\forall s_N \xrightarrow{\langle P_N, X_N, o \rangle}_N s'_N, \exists s'_S . s_S \overset{\langle P_S, X_S, f(s_N, s_S, o) \rangle}{\bullet\!\!\Longrightarrow}_S s'_S \wedge \\
& \qquad P_N \supseteq P_S \wedge \\
& \qquad X_N \subseteq X_S \wedge \\
& \qquad s'_N \sqsubseteq s'_S)
\end{aligned}
$$

The same relation can be defined as a post-fixpoint of the operator Ψ_f, which is defined as follows.

Definition 6.9 (Operator Ψ_f). *Let $S = \langle S_S, R_S, C_S, O_S, \mathcal{M}_S \rangle$ be a service template (with $\mathcal{M}_S = \langle \overline{s}_S, \rho_S, \chi_S, \tau_S \rangle$), and let $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$ be a node type (with $\mathcal{M}_N = \langle \overline{s}_N, \rho_N, \chi_N, \tau_N \rangle$). The* operator *$\Psi_f$ is defined as follows:*

$$
\begin{aligned}
\Psi_f(R) := \{ (s_N, s_S) \in R \mid & \rho_N(s_N) \supseteq \rho_S(s_S) \wedge \\
& \chi_N(s_N) \subseteq \chi_S(s_S) \wedge \\
& (\forall s_N \xrightarrow{\langle P_N, X_N, o \rangle}_N s'_N, \exists s'_S . s_S \overset{\langle P_S, X_S, f(s_N, s_S, o) \rangle}{\bullet\!\!\Longrightarrow}_S s'_S \wedge \\
& \qquad P_N \supseteq P_S \wedge \\
& \qquad X_N \subseteq X_S \wedge \\
& \qquad (s'_N, s'_S) \in R) \}
\end{aligned}
$$

Lemma 6.1. *The relation of f-simulation (viz., $\sqsubseteq_f$) is a post-fixpoint of the operator Ψ_f. Namely:*

$$\sqsubseteq_f = \Psi_f(\sqsubseteq_f).$$

Proof. The thesis follows trivially from the definitions of $\sqsubseteq_f$ (Def. 6.7) and of Ψ_f (Def. 6.9). □

In Sect. 6.4.1, we mentioned that the algorithm essentially consists of a greatest fixpoint computation on the lattices of functions $F\colon S_N \times S_S \times O_N \to 2^{W_S}$. The following definition makes formal this intuition by introducing a monotone endomap Φ on the lattice F.

Definition 6.10 (Endomap Φ). *Let $S = \langle S_S, R_S, C_S, O_S, \mathcal{M}_S \rangle$ be a service template (with $\mathcal{M}_S = \langle \overline{s}_S, \rho_S, \chi_S, \tau_S \rangle$), and let $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$ be a node type (with $\mathcal{M}_N = \langle \overline{s}_N, \rho_N, \chi_N, \tau_N \rangle$). Let also $W_S \subseteq O_S^*$ be the set of all minimal operation sequences in $\mathcal{M}_S$ (Def. 6.6). We define the* endomap Φ *on the lattice of functions $F\colon S_N \times S_S \times O_N \to 2^{W_S}$ as follows:*

$$
\begin{aligned}
\Phi(F)(s_N, s_S, o) := \{ w \in & F(s_N, s_S, o) \mid \\
& \rho_N(s_N) \supseteq \rho_S(s_S) \wedge \\
& \chi_N(s_N) \subseteq \chi_S(s_S) \wedge \\
& (\forall s_N \xrightarrow{\langle P_N, X_N, o \rangle}_N s'_N, \exists s'_S . s_S \bullet\!\!\xRightarrow{\langle P_S, X_S, w \rangle}_S s'_S \wedge \\
& \qquad P_N \supseteq P_S \wedge \\
& \qquad X_N \subseteq X_S \wedge \\
& \qquad \forall o', F(s'_N, s'_S, o') \neq \varnothing)\}
\end{aligned}
$$

Lemma 6.2. *The algorithm presented in Sect. 6.4.1 consists in computing of the greatest fixpoint of the endomap Φ.*

Proof. Consider the rules given in Sect. 6.4.1 for computing F_0 and F_{k+1}. By definition of Φ:

- F_0 is just $\Phi(\top)$, where $\top$ is the greatest element of the lattice of functions $F\colon S_N \times S_S \times O_N \to 2^{W_S}$ (i.e., a function assigning to any triple i, j, o the whole set W_s), and
- F_{n+1} is just $\Phi(F_n)$.

□

We now have all the elements needed to prove by means of coinduction that the algorithm to compute a f-simulation (see Sect. 6.4.1) is sound and complete.

Proposition 6.2. *The algorithm presented in Sect. 6.4.1 is* sound *and* complete*.*

Proof. We know that:

- The relation $\sqsubseteq_f$ is a post-fixpoint of the operator Ψ_f (by Lemma 6.1).

- The algorithm presented in Sect. 6.4.1 consists in computing of the greatest fixpoint of the endomap Φ (by Lemma 6.2).

Given the above, to prove soundness and completeness of the algorithm presented in Sect. 6.4.1 we can focus on Φ and Ψ_f, by showing how they are related. More precisely, we need to exploit Ψ_f to show that:

(a) If there is an f for which the states of a node type N are f-simulated by those of a service template S, then there is a non-empty fixpoint for the endomap Φ built on the corresponding lattice, and

(b) if Φ has a non-empty fixpoint, then it is always possible to extract from it a function f such that the states of N are f-simulated by those of S.

The above listed conditions (a) and (b) are proved by the following Lemmas 6.3 and 6.4, respectively. □

Lemma 6.3. *Let $S = \langle S_S, R_S, C_S, O_S, \mathcal{M}_S \rangle$, and let $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$. Let also $W_S \subseteq O_S^*$ be the set of all minimal operation sequences in $\mathcal{M}_S$ (Def. 6.6), and Φ be an endomap on the lattice of functions $F\colon S_N \times S_S \times O_N \to 2^{W_S}$ built as shown in Def. 6.10.*

Consider a state $s_S \in S_S$ and a state $s_N \in S_N$. If there exists a function f for which $s_N \sqsubseteq_f s_S$, then there is a non-empty fixpoint for the endomap Φ on F. In formulas, $\forall s_N \in S_N, s_S \in S_S, o \in O_N$

$$\Phi(F_f)(s_N, s_S, o) = F_f(s_N, s_S, o)$$

where $F_f(s_N, s_S, o) = \{f(s_N, s_S, o) \mid s_N \sqsubseteq_f s_S\}$.

Proof. The definition of $F_f(s_N, s_S, o)$ naturally partitions the proof in two cases, i.e. (a) $F_f(s_N, s_S, o) = \varnothing$, and (b) $F_f(s_N, s_S, o) = \{f(s_N, s_S, o)\} \neq \varnothing$.

(a) Assume that $F_f(s_N, s_S, o) = \varnothing$. By definition of Φ

$$\Phi(F_f)(s_N, s_S, o) \subseteq F_f(s_N, s_S, o).$$

Since $F_f(s_N, s_S, o) = \varnothing$

$$\Phi(F_f)(s_N, s_S, o) \subseteq \varnothing,$$

which trivially implies that

$$\Phi(F_f)(s_N, s_S, o) = \varnothing.$$

From the above, and since $F_f(s_N, s_S, o) = \varnothing$, we have that $\forall o \in O_N$

$$\Phi(F_f)(s_N, s_S, o) = F_f(s_N, s_S, o).$$

(b) On the other hand, if $F_f(s_N, s_S, o) = \{f(s_N, s_S, o)\} \neq \varnothing$, then the definition of F_f ensures that

$$s_N \sqsubseteq_f s_S.$$

By expanding the definition of $\sqsubseteq_f$ we have that

$$\begin{aligned}
&\rho_N(s_N) \supseteq \rho_S(s_S) \wedge \\
&\chi_N(s_N) \subseteq \chi_S(s_S) \wedge \\
&(\forall s_N \xrightarrow{\langle P_N, X_N, o \rangle}_N s'_N, \exists s'_S . s_S \overset{\langle P_S, X_S, f(s_N, s_S, o) \rangle}{\bullet\!\!\Longrightarrow}_S s'_S \wedge \\
&\qquad\qquad P_N \supseteq P_S \wedge \\
&\qquad\qquad X_N \subseteq X_S \wedge \\
&\qquad\qquad s'_N \sqsubseteq_f s'_S.
\end{aligned}$$

Since $s'_N \sqsubseteq_f s'_S \Rightarrow \forall o', F_f(p', q', o') = \{f(p', q', o')\} \neq \varnothing$, the above can be rewritten as follows:

$$\begin{aligned}
&\rho_N(s_N) \supseteq \rho_S(s_S) \wedge \\
&\chi_N(s_N) \subseteq \chi_S(s_S) \wedge \\
&(\forall s_N \xrightarrow{\langle P_N, X_N, o \rangle}_N s'_N, \exists s'_S . s_S \overset{\langle P_S, X_S, f(s_N, s_S, o) \rangle}{\bullet\!\!\Longrightarrow}_S s'_S \wedge \\
&\qquad\qquad P_N \supseteq P_S \wedge \\
&\qquad\qquad X_N \subseteq X_S \wedge \\
&\qquad\qquad \forall o' \in O_N, F_f(s'_N, s'_S, o') \neq \varnothing.
\end{aligned}$$

The above predicate is proved to be true, and this means that we can write the following equivalence (for every $o \in O_N$):

$$\begin{aligned}
\{f(s_N, s_S, o)\} = \{f(s_N, s_S, o) \mid \\
&\rho_N(s_N) \supseteq \rho_S(s_S) \wedge \\
&\chi_N(s_N) \subseteq \chi_S(s_S) \wedge \\
&(\forall s_N \xrightarrow{\langle P_N, X_N, o \rangle}_N s'_N, \exists s'_S . s_S \overset{\langle P_S, X_S, f(s_N, s_S, o) \rangle}{\bullet\!\!\Longrightarrow}_S s'_S \wedge \\
&\qquad\qquad P_N \supseteq P_S \wedge \\
&\qquad\qquad X_N \subseteq X_S \wedge \\
&\qquad\qquad \forall o', F_f(p', q', o') \neq \varnothing\},
\end{aligned}$$

which by definition of Φ (Def. 6.10) means that (for every $o \in O_N$)

$$\{f(s_N, s_S, o)\} = \Phi(F_f)(s_N, s_S, o)$$

Finally, since $F_f = \{f(s_N, s_S, o)\}$, we obtain that (for every $o \in O_N$)

$$F_f = \Phi(F_f)(s_N, s_S, o)$$

□

Lemma 6.4. *Let $S = \langle S_S, R_S, C_S, O_S, \mathcal{M}_S \rangle$, and let $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$. Let also Ψ_f be the operator defined in Def. 6.9, $W_S \subseteq O_S^*$ be the set of all minimal operation sequences in $\mathcal{M}_S$ (Def. 6.6), and Φ be an endomap on the lattice of functions $F\colon S_N \times S_S \times O_N \to 2^{W_S}$ built as shown in Def. 6.10.*

If Φ has a non-empty fixpoint, then it is possible to extract from it a function f for which $s_N \sqsubseteq_f s_S$ (with $s_N \in S_N$ and $s_S \in S_S$). Formally, assuming that

(i) $\forall s_N \in S_N, s_S \in S_S, o \in O_N . \Phi(F)(s_N, s_S, o) = F(s_N, s_S, o)$, *and*

(ii) $f_F : S_N \times S_S \times O_N \to O_S^*$
such that
$\forall s_N \in S_N, s_S \in S_S . (\forall o \in O_N . F(s_N, s_S, o) = \varnothing) \Rightarrow f(s_N, s_S, o) \in F(s_N, s_S, o)$,

we have that

$$\sqsubseteq_F \subseteq \Psi_{f_F}(\sqsubseteq_F)$$

where $s_N \sqsubseteq_F s_S := \forall o \in O_N, F(s_N, s_S, o) \neq \varnothing$.

Proof. Consider a state $s_N \in S_N$ and a state $s_S \in S_S$, and suppose that

$$s_N \sqsubseteq_F s_S.$$

By definition of $\sqsubseteq_F$, we have that

$$\forall o \in O_N . F(s_N, s_S, o) \neq \varnothing.$$

The above, along with the hypoteses on f, implies that

$$\forall o \in O_N . f(s_N, s_S, o) \in F(s_N, s_S, o) \subseteq \Phi(F)(s_N, s_S, o).$$

By definition of Φ, the above can be rewritten as follows

$$\begin{aligned}
\forall o \in O_N . \rho_N(s_N) \supseteq \rho_S(s_S) \wedge \\
\chi_N(s_N) \subseteq \chi_S(s_S) \wedge \\
(\forall s_N \xrightarrow{\langle P_N, X_N, o \rangle}_N s'_N, \exists s'_S \in S_S . s_S \bullet\!\!\xRightarrow{\langle P_S, X_S, f(s_N, s_S, o) \rangle}_S s'_S \wedge \\
P_N \supseteq P_S \wedge \\
X_N \subseteq X_S \wedge \\
\forall o', F(s'_N, s'_S, o') \neq \varnothing),
\end{aligned}$$

which, by definition of $\sqsubseteq_F$, can be in turn rewritten as follows

$$
\begin{aligned}
\forall o \in O_N . \rho_N(s_N) \supseteq \rho_S(s_S) \wedge \\
\chi_N(s_N) \subseteq \chi_S(s_S) \wedge \\
(\forall s_N \xrightarrow{\langle P_N, X_N, o\rangle}_N s'_N, \exists s'_S \in S_S . s_S \overset{\langle P_S, X_S, f(s_N, s_S, o)\rangle}{\bullet\!\!\Longrightarrow}_S s'_S \wedge \\
P_N \supseteq P_S \wedge \\
X_N \subseteq X_S \wedge \\
s'_N \sqsubseteq_F s'_S).
\end{aligned}
$$

From the above, and because of the definition of Ψ_{f_F}, we have that

$$(s_N, s_S) \in \Psi_{f_F}(\sqsubseteq_F),$$

from which it follows the thesis we wanted to prove. $\square$

Chapter 7

Fault-aware modelling and analysis of application management

In Chapter 5 we have shown how the management behaviour of topology nodes can be modelled by *management protocols*, specified as finite state machines whose states and transitions are associated with conditions defining the consistency of the states of a node and constraining the executability of management operations. Such conditions are defined on the requirements of a node, and each requirement of a node has to be fulfilled by a capability of another node. As a consequence, the management behaviour of a composite application can be easily derived by composing the management protocols of its nodes according to the dependencies defined in its topology.

The above does not deal with the potential occurrence of faults, which however must be considered when managing complex composite applications [43]. Indeed, an application component may be affected by faults caused by other components on which it relies (e.g., a component is shutdown or uninstalled while another component is relying on its capabilities).

In Sect. 7.2 we propose a *fault-aware* extension of management protocols, to permit modelling how nodes behave when faults occur. We also illustrate how to analyse and automate the management of composite applications in a fault-resilient manner. Namely, we show how the fault-aware management behaviour of a composite application can be determined by composing the protocols of its nodes according to the application's topology. We then describe how to determine whether a plan orchestrating the management of an application is valid, which are its effects (e.g., which capabilities are available after executing it, or whether it may generate faults while being executed), and how this also permits finding management plans from given application configurations to achieve specific goals.

Notice that, even if the components of an application are described by fault-aware management protocols, the actual behaviour of such components may differ from the described one (e.g., because of a non-deterministic

bug [64]). In Sect. 7.4 we show how the unexpected behaviour of a component can be modelled by automatically completing its management protocols, and how this permits analysing the (worst possible) effects of a misbehaving component on the rest of the application. We also illustrate a way to hard recover applications that are stuck because a fault was not properly handled, or because of a misbehaving component (see Sect. 7.5).

Finally, to illustrate the feasibilty of our approach, in Sect. 7.6 we describe a proof-of-concept, web-based application that permits editing fault-aware management protocols in TOSCA applications, and which permits analysing the management of such applications. In Sect. 7.7 we also show how to exploit our proof-of-concept implementation to validate and automate the fault-aware management of the *Thinking* case study (previously discussed in Sect. 5.5).

7.1 Motivating scenario

Consider a toy application composed by a web-based *Frontend* and a *Backend*, both deployed on an *Apache* server, which in turn is installed on a *Debian* operating system. Fig. 7.1 illustrates the topology of such application, according to the TOSCA graphical notation introduced by Winery [78].

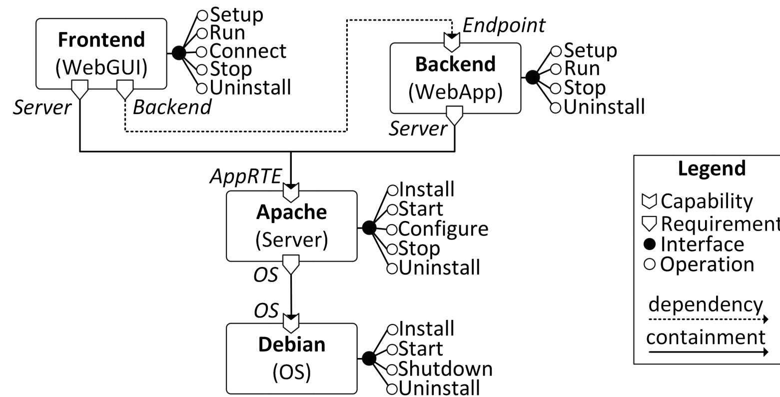


FIGURE 7.1: Motivating example.

Each inter-node dependency is explicitly represented by a relationship connecting a node's requirement with another node's capability (e.g., the *Server* requirements of *Frontend* and *Backend* are connected with the *AppRTE* capability of *Apache*). A relationship can represent a "vertical" containment dependency, specifying that a component is contained in another (e.g., *Apache* is installed on *Debian*), or an "horizontal" dependency, specifying that a component just requires another (without stating that the former is contained in the latter — e.g., *Frontend* must connect to *Backend*'s *Endpoint* to work properly).

Suppose for instance that all nodes have been deployed, started, and properly connected each other (i.e., all components are in their *running* state). What happens if the *Stop* operation of *Backend* is executed? The *Backend* application component is stopped, and this generates a fault in the *Frontend*, which becomes unable to serve requests to its clients, simply because the connection with *Backend* is not working any more. Furthermore, even if *Backend* is re-started, the *Frontend* has to re-connect to the *Backend*.

Even worse is the case when a node presents an unexpected behaviour. Suppose again that the application is up and running, and that the *Apache* server unexpectedly crashes. Such a crash results in faulting also the nodes contained in *Apache* (viz., *Frontend* and *Backend*), which are suddenly killed, and potentially enter in an inconsistent state that makes them unusable from there onwards.

Both the above mentioned cases fail because a node stops providing its capabilities while other nodes are relying on them to continue to work. In the first case this happens because of the invocation of a management operation that stops a node while other nodes are depending on it. In the second case a node unpredictably fails[1].

7.2 Fault-aware application management protocols

7.2.1 Definition of fault-aware management protocols

We hereby define a *fault-aware* extension of *management protocols* (which have been first introduced in Chapter 5), to permit describing how N reacts when it is in a state assuming some requirements to be satisfied, and some other node(s) stop(s) providing the capabilities satisfying such requirements. We introduce a new transition relation φ to model the explicit fault handling of N, i.e. how N changes its state from s to s' when some of the requirements it assumes in s stop being satisfied.

Definition 7.1 (Fault-aware management protocol). *Let $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$ be a node, where S_N, R_N, C_N, and O_N are the finite sets of its states, requirements, capabilities, and management operations. $\mathcal{M}_N = \langle \overline{s}_N, \rho_N, \chi_N, \tau_N, \varphi_N \rangle$ is a finite state machine defining the* fault-aware management protocol *of N, where:*

- *$\overline{s}_N \in S_N$ is the initial state,*
- *$\rho_N : S_N \to 2^{R_N}$ is a function indicating which requirements must hold in each state $s \in S_N$,*

[1] Misbehaving components can be detected via monitoring (e.g., by exploiting watchdogs or heartbeat services). We shall not deepen into details, as component monitoring is outside of the scope of this thesis.

- $\chi_N : S_N \rightarrow 2^{C_N}$ *is a function indicating which capabilities of* N *are offered in a state* $s \in S_N$*,*
- $\tau_N \subseteq S_N \times 2^{R_N} \times 2^{C_N} \times O_N \times S_N$ *is a set of quintuples modelling the transition relation (i.e.,* $\langle s, P, X, o, s' \rangle \in \tau_N$ *denotes that in state* s*, and if condition* P *holds,* o *is executable and leads to state* s' *— by maintaining the capability in* X *during the transition), and*
- $\varphi_N \subseteq S_N \times 2^{R_N} \times S_N$ *is a set of triples modelling the explicit fault handling for a node (i.e.,* $\langle s, H, s' \rangle \in \varphi_N$ *denotes that the node will change its state from* s *to* s' *if the requirements in* H *stop being satisfied).*

Example 7.1. Fig. 7.2 shows the fault-aware management protocols of the nodes composing our motivating scenario (where thicker arrows represent τ, and lighter arrows represent φ).

Consider for instance the fault-aware management protocol $\mathcal{M}_{Apache}$, which describes the behaviour of the *Apache* node. In its initial state (*NotInstalled*) *Apache* does not require nor provide anything. In the *Installed* and *Started* states it instead assumes the *OS* requirement to (continue to) be satisfied. If the *OS* requirement is faulted, then *Apache* returns to its initial state (thus requiring to be installed and started again). The *Started* state is the only one where *Apache* concretely provides its *AppRTE* capability. Finally, the protocol specifies that all *Apache*'s operations can be performed only if the *OS* requirement is satisfied.

Consider now the fault-aware management protocol $\mathcal{M}_{Backend}$, which describes the behaviour of the *Backend* node. The description is "incomplete": When *Backend* is *Installed* or *Running*, it assumes the capability satisfying its *Server* requirement to (continue to) be provided. It however does not describe what happens if such capability stops being provided. A way to automatically complete fault-aware management protocols (by adding transitions for all unhandled faults) is provided in Sect. 7.2.3. □

7.2.2 Characterising fault-aware management protocols

We now show how to formally define the constraints to ensure determinism and well-formedness of fault-aware management protocols. To do it, we shall extend the analogous constraints we discussed in Sect. 5.2.2.

A management protocol is *deterministic* if (i) management operations have deterministic effects when applied in a state (viz., it is not possible to have a state with two outgoing transitions corresponding to the same operation and leading to different states). Additionally, (ii) fault handling transitions have to be uniquely determined by the sets of requirements that are no more satisfied.

Definition 7.2 (Determinism of fault-aware management protocols)**.** *Let* $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$ *be a node, and let* $\mathcal{M}_N = \langle \overline{s}_N, \rho_N, \chi_N, \tau_N, \varphi_N \rangle$ *be its management protocol.* $\mathcal{M}_N$ *is* deterministic *iff*

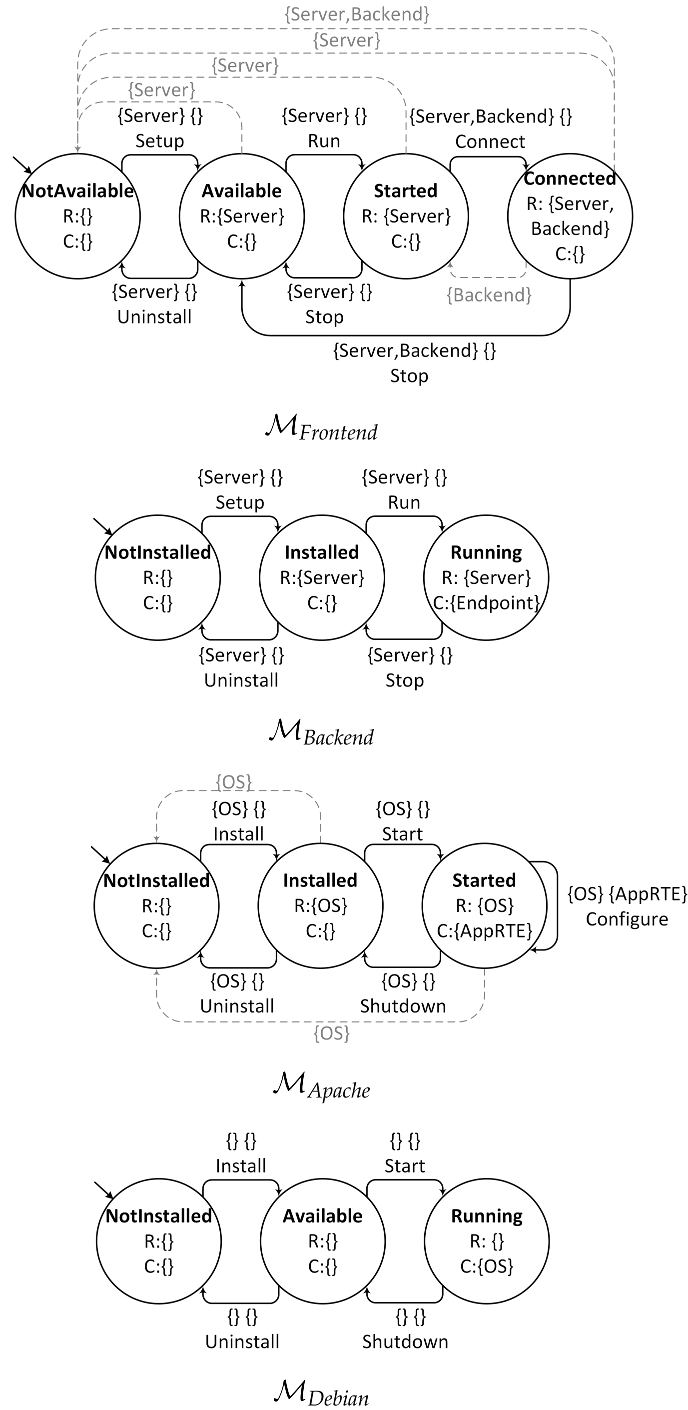


FIGURE 7.2: Examples of fault-aware management protocols.

(i) $\forall\langle s_1, P_1, X_1, o_1, s_1'\rangle, \langle s_2, P_2, X_2, o_2, s_2'\rangle \in \tau_N$:
$$(s_1 = s_2 \wedge o_1 = o_2) \Rightarrow s_1' = s_2'.$$

(ii) $\forall\langle s_1, H_1, s_1'\rangle, \langle s_2, H_2, s_2'\rangle \in \varphi_N$:
$$\big(s_1 = s_2 \wedge H_1 = H_2\big) \Rightarrow s_1' = s_2'.$$

It can be trivially verified that all protocols in Fig. 7.2 are deterministic since there is no pair of transitions which start from the same source state and leading to different states by (i) applying the same operation or (ii) handling the same faulted requirements.

A management protocol is also *well-formed* if the conditions on requirements of each transition t are consistent with the source and target states. This means that (i) the requirements assumed to hold in the source state of t, as well as those assumed to hold in its target state, must be assumed to hold also during the transition, to avoid inconsistencies. (ii) Faults can only affect requirements that are assumed in a state, and such faults should lead to states where faulted requirements are no more assumed and where no additional capabilities are provided.

Definition 7.3 (Well-formedness of fault-aware management protocols)**.** *Let* $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N\rangle$ *be a node, and let* $\mathcal{M}_N = \langle \overline{s}_N, \rho_N, \chi_N, \tau_N, \varphi_N\rangle$ *be its management protocol.* $\mathcal{M}_N$ *is* well-formed *iff*

(i) $\forall\langle s, P, X, o, s'\rangle \in \tau_N$:
$$\rho_N(s) \cup \rho_N(s') \subseteq P \;\wedge\; X \subseteq \chi_N(s) \cap \chi_N(s'), \text{ and}$$

(ii) $\forall\langle s, H, s'\rangle \in \varphi_N$:
$$\varnothing \neq H \subseteq \rho_N(s) \;\wedge\; \rho_N(s') \subseteq \rho_N(s) - H \;\wedge\; \chi_N(s') \subseteq \chi_N(s)$$

It can be trivially verified that all protocols in Fig. 7.2 are well-formed, since (i) whatever transition $t \in \tau_*$ we consider, the set of requirements needed to fire t is the union of the requirements assumed in the source and target states of t, and since (ii) whatever transition $f \in \varphi_*$ we take, the faulted requirements it handles are a subset of the requirements in the source state of f and its target state assumes a set of requirements which is exactly given by removing the faulted requirements handled by f from the requirements assumed in its source state.

Finally, as we will see in Sect. 7.3, faults are not going to be propagated synchronously, i.e. when a capability is removed, the nodes assuming the requirements satisfied by such capability eventually detect the removal, but in the meanwhile other capabilities might disappear (thus making other requirements unsatisfied). For this reason, (the fault handling in) management protocols should be *race-free*, which means that the simultaneous removal of multiple requirements should have the same effect on a node as

any sequential removal of the same requirements, if no operations are executed on the node in the meantime. More precisely, (i) if φ permits transiting from state s to state s', then there must be a φ transition which handles all removed requirements, (ii) if a set of removed requirements is handled in a state, then all its subsets have also to be handled in the same state, (iii) if the removal of two sets of requirements is handled in a state, then the removal of their union has to be handled in the same state, (iv) the φ relation has to be transitive, and (v) if a fault can be handled in a state s, then the same fault has to be handled in all states that can be reached from s and that have not yet handled it.

Definition 7.4 (Race-freedom of fault-aware management protocols)**.** *Let $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$ be a node, and let $\mathcal{M}_N = \langle \overline{s}_N, \rho_N, \chi_N, \tau_N, \varphi_N \rangle$ be its management protocol. $\mathcal{M}_N$ is* race-free *iff*

(i) $\forall \langle s, H, s' \rangle \in \varphi_N : \langle s, \rho(s) - \rho(s'), s' \rangle \in \varphi_N$.

(ii) $\forall \langle s, H, s' \rangle \in \varphi_N : \forall \varnothing \neq R \subset H \Rightarrow \exists \langle s, R, s'' \rangle \in \varphi_N$.

(iii) $\forall \langle s, H', s' \rangle, \langle s, H'', s'' \rangle \in \varphi_N : \exists \langle s, H' \cup H'', s''' \rangle \in \varphi_N$.

(iv) $\forall \langle s, H', s' \rangle, \langle s', H'', s'' \rangle \in \varphi_N : \langle s, H' \cup H'', s'' \rangle \in \varphi_N$.

(v) $\forall \langle s, H', s' \rangle, \langle s, H'', s'' \rangle \in \varphi_N : H'' \subseteq \rho(s') \Rightarrow \exists \langle s', H'', s''' \rangle \in \varphi_N$

It is easy to check that all protocols in Fig. 7.2 are race-free.

In the following we assume fault-aware management protocols to be well-formed, deterministic, and race-free. Notice that partially specified φ relations can be automatically completed by applying the rules in Def. 7.4, therefore assuming fault-aware management protocols to be race-free does not require additional effort when modelling management protocols.

7.2.3 Completing fault-aware management protocols

As illustrated by Example 7.1, the management protocol of a node may leave unspecified how the component will behave in case some requirements stop being fulfilled in some states. To explicitly model that, management protocols can be completed by adding transitions for all unhandled faults, all leading to a "sink" state $s_{\lightning}$ (that requires and provides nothing)[2].

Definition 7.5 (Completing fault-aware management protocols)**.** *Let $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$ be a node, where $\mathcal{M}_N = \langle \overline{s}_N, \rho_N, \chi_N, \tau_N, \varphi_N \rangle$ is its fault-aware management protocol. The management protocol $\mathcal{M}_N$ can be* completed *by replacing S_N and φ_N with:*

- *$S'_N = S_N \cup \{s_{\lightning}\}$, with $s_{\lightning} \notin S_N$ and $\rho(s_{\lightning}) = \chi(s_{\lightning}) = \varnothing$, and*

[2]It is easy to prove that the proposed completion preserves the determinism, well-formedness, and race-freedom of a fault-aware management protocol.

- $\varphi'_N = \varphi_N \cup \{\langle s, H, s_{\lightning} \rangle \mid s \in S_N \wedge \varnothing \neq H \subseteq \rho(s) \wedge \nexists \langle s, H, s' \rangle \in \varphi_N\}$.

In the following we will assume fault-aware management protocols to be automatically completed as defined above. Intuitively speaking, this will ensure that composite applications will always be able to propagate whatever fault of their nodes, as from each state of a node N it will always be possible to react to the removal of any of its requirements (through one of the transitions in the original φ_N, or through those introduced in φ'_N).

Example 7.2. The completion of the management protocol $\mathcal{M}_{Backend}$ (Fig. 7.2) is shown in Fig. 7.3: We add a "sink state" *Backend*$_{\lightning}$, and two transitions reacting to the unsatisfaction of the *Server* requirement when *Backend* is in its *Installed* or *Running* states.

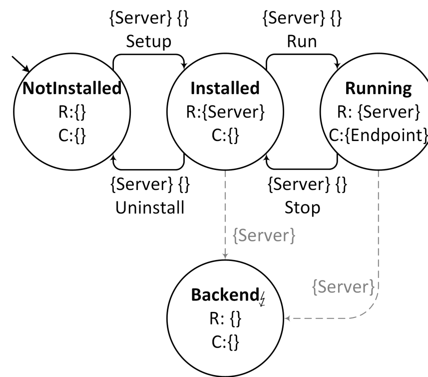


FIGURE 7.3: Example of completed management protocol.

The extension of the other management protocols in Fig. 7.2 is even simpler: Since they handle all potential faults, their extension only consists in adding a sink state to each of them (i.e., *Frontend*$_{\lightning}$ is added to the *Frontend*'s states, while *Apache*$_{\lightning}$ and *Debian*$_{\lightning}$ are added to those of *Apache* and *Debian*, respectively). □

7.3 Analysis of fault-aware management protocols

In this section we generalise the analyses presented in Sect. 5.3 so as to take into account the potential occurrence of faults. Namely, we first show how to infer the fault-aware management behaviour of an application by composing the fault-aware management protocols of its components, and then we describe different analyses that can be performed on such behaviour (e.g., checking the validity of a plan, determining its effects or whether it generates faults while being executed, or discovering plans that permit reaching certain system configurations).

In doing so, we exploit the same shorthand notation as in Sect. 5.3. For the convenience of readers, we recall such notation below.

Notation 7.1. *We denote with $A = \langle T, b \rangle$ a generic composite application, where T is the finite set of nodes in the application topology*[3]*, and where the connection among nodes is described by a (total)* binding *function*

$$b : \bigcup_{N \in T} R_N \rightarrow \bigcup_{N \in T} C_N$$

associating each node's requirement with the capability satisfying it.

Let G be a global state of A (see Def. 5.4). We denote with $\rho(G)$ the set of requirements that are assumed to hold by the nodes in T when A is in G, with $\chi(G)$ the set of capabilities that are provided by such nodes in G, and with $b(R)$ the set of capabilities bound to the requirements in R. Formally:

- $\rho(G) = \bigcup_{N \in T} \{\rho_N(s) \mid s \in G \wedge s \in S_N\}$,
- $\chi(G) = \bigcup_{N \in T} \{\chi_N(s) \mid s \in G \wedge s \in S_N\}$, *and*
- $b(R) = \bigcup_{r \in R} \{b(r)\}$.

7.3.1 Fault-aware management behaviour

Since A defines a composition of the nodes in T that coordinate through the binding b among requirements and capabilities, we model the behaviour of A by simply composing the management protocols of the nodes in T.

First, we generalise the notion of global state of A by introducing pending faults. According to Def. 5.4, the global state of an application A is a set G containing the current state of each of its nodes. We define a function F to denote the set of pending faults in G, which are the requirements assumed in G while the corresponding capabilities are not provided[4].

Definition 7.6 (Pending faults)**.** *Let $A = \langle T, b \rangle$ be a composite application, and let $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$. Let also G be a global state of A (Def. 5.4), i.e. G is a set of states such that:*

$$G \subseteq \bigcup_{N \in T} S_N \ \wedge \ \forall N \in T \colon \exists! s \in G \cap S_N.$$

The set $F(G)$ of pending faults *in G is defined as follows:*

$$F(G) = \{r \in \rho(G) \mid b(r) \notin \chi(G)\}.$$

The management behaviour of a composite application A is defined by a labelled transition system over its global states (Def. 5.5). We hereby generalise such labelled transition system, by defining two simple inference rules, *(op)* for operation execution and *(fault)* for fault propagation.

[3]For simplicity, and without loss of generality, we assume that, given two nodes in a topology, the names of states, requirements, capabilities, and operations are disjoint.

[4]Such faults are considered to be "pending", as the transitions handling them have not been fired yet.

Definition 7.7 (Fault-aware management behaviour of a composite application)**.** *Let $A = \langle T, b \rangle$ be a composite application, and let $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$ with $\mathcal{M}_N = \langle \overline{s}_N, \rho_N, \chi_N, \tau_N, \varphi_N \rangle$. The* fault-aware management behaviour *of A is modelled by a labelled transition system whose configurations are the global states of A, and whose transition relation is defined by the following inference rules:*

$$\frac{s \in G \quad \langle s, P, X, o, s' \rangle \in \tau_N \quad F(G) = \varnothing \quad b(P) \subseteq \chi(G)}{G \xrightarrow{o} (G - \{s\}) \cup \{s'\}} \; \textit{(op)}$$

$$\frac{s \in G \quad \langle s, H, s' \rangle \in \varphi_N \quad H \subseteq F(G)}{G \xrightarrow{\bot} (G - \{s\}) \cup \{s'\}} \; \textit{(fault)}$$

The *(op)* rule[5] defines how the global state of A is updated when a node N performs a transition $\langle s, P, X, o, s' \rangle \in \tau_N$. Such transition can be performed when there are no pending faults (viz., $F(G) = \varnothing$), and the requirements needed to perform the transition are satisfied in G (viz., $b(P) \subseteq \chi(G)$). As a result, the global state G is updated with the new state of N (viz., $G' = (G - \{s\}) \cup \{s'\}$), potentially triggering faults to be handled (if $F(G') \neq \varnothing$).

The *(fault)* rule instead models fault propagation. Such rule defines how the global state G of an application A is updated when executing a fault handling transition $\langle s, H, s' \rangle$ of a node N. Such transition can be executed if the faults it handles are pending in G (viz., $H \subseteq F(G)$), and its effects on the whole application A are the following: The state of N is updated (viz., $G' = (G - \{s\}) \cup \{s'\}$), novel faults may be triggered, while the faults in H are not pending any more[6].

7.3.2 Fault-aware analysis of application management

The management behaviour defined in Def. 7.7 permits analysing and automating the management of a composite application. For instance, we can easily generalise the notion of *validity* for sequences of management operations (Def. 5.7) and for management plans (Def. 5.8) as follows.

Definition 7.8 (Valid plan)**.** *Let $A = \langle T, b \rangle$ be a composite application. The sequence $o_1 o_2 ... o_n$ of management operations in A is* valid *in a global state G_0 of A iff*

$$\exists G_1, G_2, ... G_n : G_0 \xmapsto{o_1} G_1 \xmapsto{o_2} G_2 \xmapsto{o_3} \ldots \xmapsto{o_n} G_n$$

[5]It is easy to see that the *(op)* is a generalisation of the rule in Def. 5.5, whose only additional constraint is the absence of pending faults.

[6]The faults in H are not pending any more since $\rho_N(s') \subseteq \rho_N(s) - H$ by well-formedness of management protocols (Def. 7.3).

where

$$\frac{G \xrightarrow{o} G'}{G \overset{o}{\mapsto} G'} \qquad \frac{G \overset{o}{\mapsto} G' \quad G' \xrightarrow{\bot} G''}{G \overset{o}{\mapsto} G''}$$

A plan P orchestrating the management operations in A is valid *in G_0 iff all its sequential traces are valid in G_0.*

Example 7.3. Consider the workflow in Fig. 7.4, which permits restarting the *Backend* and *Frontend* components of our motivating application (Figs. 7.1 and 7.2). Suppose also that the application is in the following global state: *Debian* is *Running*, *Apache* is *Started*, *Backend* is *Running*, and *Frontend* is *Connected*. It is easy to check that the plan is valid in the considered global state since both its sequential traces are valid in such global state.

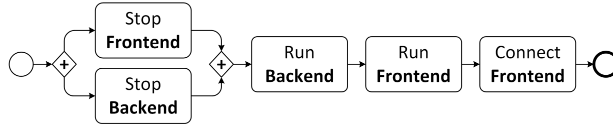


FIGURE 7.4: Example of valid plan.

Consider, for instance, the sequential trace performing *Backend*'s *Stop* before *Frontend*'s *Stop*. Fig. 7.5.(b) shows the validity of such a sequential trace by illustrating the evolution of the application's global state. □

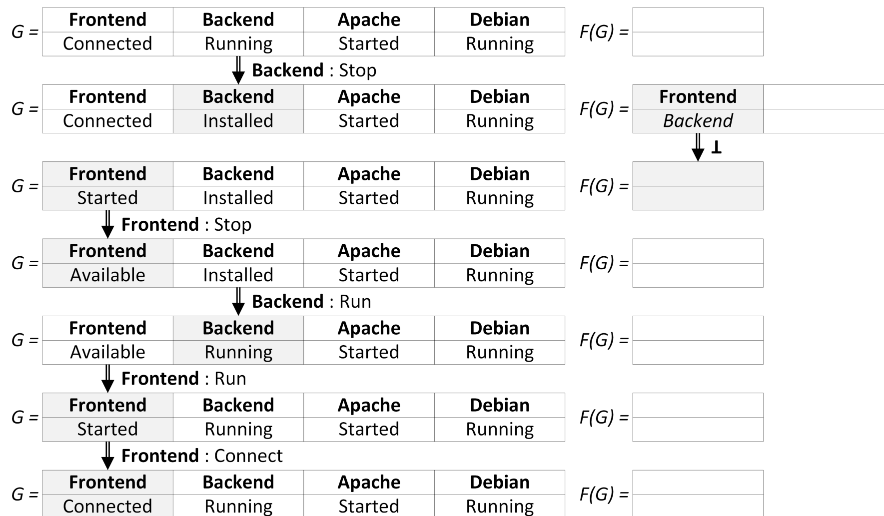


FIGURE 7.5: Example of valid sequence of operations.

Notice that Def. 7.8 generalises the notion of valid plan given in Def. 5.8 by permitting faults to happen within a valid plan. In case we do not want faults to happen within a plan, we should require such plan to be *fault-free*.

Definition 7.9 (Fault-free plan)**.** *Let A be a composite application, and let G be a global state. Let also P be a valid plan for A in G. P is also* fault-free *if no $\bot$-transition is performed in any of its sequential traces.*

Remark 7.1. It can be trivially checked that the notion of *fault-freeness* for fault-aware management protocols (Def. 7.9) is analogous to that of *validity* for "plain" management protocols (see Def. 5.8 in Chapter 5). □

The modelling introduced above can be exploited for various other purposes besides checking whether a plan is valid or fault-free. For instance, we may be interested in checking whether a plan is determinstic, which are the effects of a plan, whether there exists a plan that permit reaching a desired application configuration, or whether an application is softly resettable. By exploiting the updated notion of validity (Def. 7.8), we can perform all such analyses as illustrated in Sect. 5.3.

7.4 Modelling and analysing "the unexpected"

The analysis described in Sect. 7.3 assumes that each application component behaves accordingly to its specified management protocol, thus not taking into account components that behave unexpectedly because of mismatches between their modelled and actual behaviour (e.g., because of a bug). We hereby illustrate a way to deal with such a kind of situations.

The unexpected behaviour of a component can be modelled by automatically completing its management protocol by adding a "crash" operation $\lightning$ that leads the node to the sink state $s_{\lightning}$.

Definition 7.10 (Fault-aware management protocols with unexpected behaviour)**.** *Let $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$ be a node, where $\mathcal{M}_N = \langle \overline{s}_N, \rho_N, \chi_N, \tau_N, \varphi_N \rangle$ is its fault-aware management protocol. The fault-aware management protocol of N can be extended to include* unexpected behaviour *by replacing O_N and τ_N with:*

- $O'_N = O_N \cup \{\lightning\}$*, and*
- $\tau'_N = \tau_N \cup \{\langle s, \rho(s), \lightning, s_{\lightning} \rangle \mid s \in S_N\}$[7].

The $\lightning$ operation, combined with the analyses presented in Sect. 7.3.2, permits analysing the management behaviour of a composite application also in presence of misbehaving components: Indeed, the possible unexpected behaviour of a node is modelled by $\lightning$ transitions which lead the nodes to their sink state $s_{\lightning}$, where we (pessimistically) assume that the node is not offering any capability any more. This permits us to analyse the (worst possible) effects of a misbehaving node on the rest of the application by simply observing how the global state of the application changes.

[7] $\lightning$ transitions can be fired only if the requirements in $\rho(s)$ are satisfied so as to ensure the well-formedness of fault-aware management protocols (Def. 7.3). Notice that this is not a restriction since such requirements are satisfied in s (by Defs. 7.1 and 7.5).

Example 7.4. Consider the management protocol of *Backend* (Fig. 7.2), extended by adding $Backend_{\lightning}$ as illustrated in Example 7.2. The extension described in Def. 7.5 simply consists in adding "crash" transitions starting from *NotInstalled*, *Installed*, and *Running*, and leading to $Backend_{\lightning}$ (Fig. 7.6). The management protocols of *Frontend*, *Apache* and *Debian* can be extended analogously.

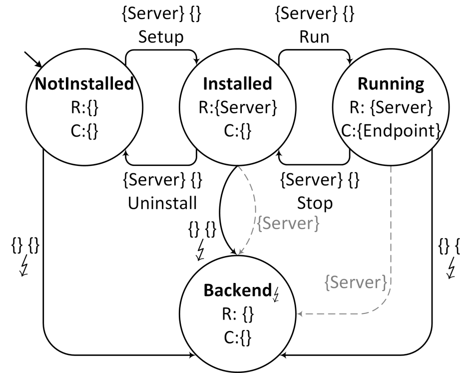


FIGURE 7.6: Example of a management protocol including unexpected behaviour.

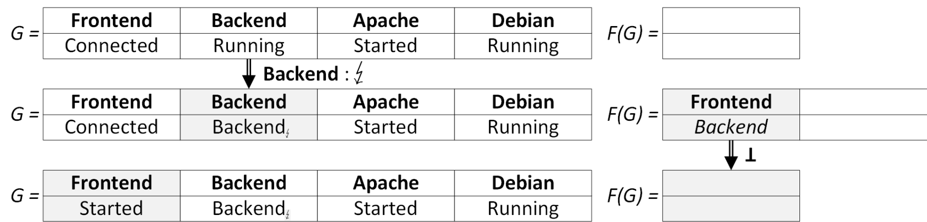


FIGURE 7.7: Example of fault injection and subsequent global state update.

The above extension permits, for instance, determining the effects of a "crashing" *Backend* when the whole application is up and running. As illustrated by Fig. 7.7, by invoking the $\lightning$ operation of *Backend*, the global state is changed by updating the state of *Backend*, and by filling the set of pending faults with the *Backend* requirement of *Frontend* (since it is assumed and connected to the *Endpoint* capability of *Backend*, which is no more provided). The pending fault is then consumed by a $\bot$-transition, which updates the state of *Frontend*. □

More interestingly, we may wish to recover an application having a component that is behaving unexpectedly. More precisely, from the global state reached after injecting a failure (by invoking the "crash" operation $\lightning$), we may wish to find a "recovery" plan whose execution permits reaching a given recovery goal (e.g., the global state in which the failure was injected). Notice that, such a recovery plan cannot be determined by simply visiting of the graph associated with the labelled transition system modelling the management behaviour of an application, as the faulted node is stuck in its sink state (since no transition outgoes from such state).

7.5 Hard recovery

Recovery plans can be generated automatically, and the underlying idea is quite simple. When a node N is stuck[8] in state $s_{\lightning}$, it can be "hard reset" by the node N' in which it is contained (i.e., by the node in which it is installed or deployed). More precisely, by resetting the container node N', all nodes it contains (among which we have the stuck node N) are forcibly reset to their initial state and can be re-installed and started to return up and running.

We hereby show how to automatically extend the modelling of an application so that hard recovery plans can be naturally determined with a visit of the graph associated with the transition system defined by the (extended) management behaviour of the application.

7.5.1 Enabling hard recovery

Consider a generic composite application $A = \langle T, b \rangle$, where T is the finite set of nodes in the application topology, and where b is the total function associating each node's requirement with the capability satisfying it. Our objective is to enforce the hard reset of a node $N \in T$ stuck in its sink state $s_{\lightning}$, by restarting the node in which N is contained. This can be naturally modelled with fault-aware management protocols, provided that the topology is extended by (i) explictly representing node containment, and (ii) updating management protocols to permit forcibly resetting container nodes whenever needed.

Notation 7.2. *To simplify notation, we denote with container(N) the node in which N is contained*[9] *(e.g., in our motivating example, container(Frontend) = Apache). If N is not contained in any other node, then container(N) is not defined (e.g., in our motivating example, container(Debian) = $\perp$).*

To explicitly represent node containment, we adapt the application A into an application $A' = \langle T', b' \rangle$, where T' and b' are built as follows:

- Each node N in the topology T is equipped with a capability $\mathsf{alive}^{\mathsf{c}}_N$ whose purpose is to allow N to witness that it continues to be available to the nodes it contains. If N is contained in another node N', then N is also equipped with a requirement $\mathsf{alive}^{\mathsf{r}}_N$ whose purpose is to permit checking whether its container N' continues to be available. Formally:

$$T' = \{\langle S_N, R'_N, C'_N, O_N, \mathcal{M}_N \rangle \mid \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle \in T\}$$

[8] In general, hard recovery can be exploited for recovering a desired global state whenever a node is stuck in its sink state (e.g., because a fault is handled by a transition that leads to the node's sink state, or because the node is behaving unexpectedly and the $\lightning$ transition has been fired).

[9] We assume that each node $N \in T$ can be contained in at most another node.

where

$$R'_N = \begin{cases} R_N \cup \{\mathsf{alive}^{\mathsf{r}}_N\} & \text{if container}(N) \neq \bot \\ R_N & \text{otherwise} \end{cases}$$

and

$$C'_N = C_N \cup \{\mathsf{alive}^{\mathsf{c}}_N\}.$$

- The function b is updated by adding the bindings among the newly introduced requirements and capabilities (viz., each requirement $\mathsf{alive}^{\mathsf{r}}_N$ is bound to the capability $\mathsf{alive}^{\mathsf{c}}_{N'}$, where N' is the node containing N). Formally:

$$b'(r) = \begin{cases} \mathsf{alive}^{\mathsf{c}}_{N'} & \text{if } r = \mathsf{alive}^{\mathsf{r}}_N \text{ and container}(N) = N' \\ b(r) & \text{otherwise} \end{cases}$$

Example 7.5. In our motivating scenario (Fig. 7.1) the above construction results in updating the application topology as illustrated in Fig. 7.8. All nodes (but *Debian*) are equipped with *alive* requirements and capabilities. *Debian* is only provided with an *alive* capability. Then, since *Frontend* and *Backend* are contained in *Apache*, the *alive* requirements of *Frontend* and *Backend* are connected with the *alive* capability of *Apache*. Additionally, since *Apache* is contained in *Debian*, the *alive* requirement of *Apache* is connected to the *alive* capability of *Debian*. □

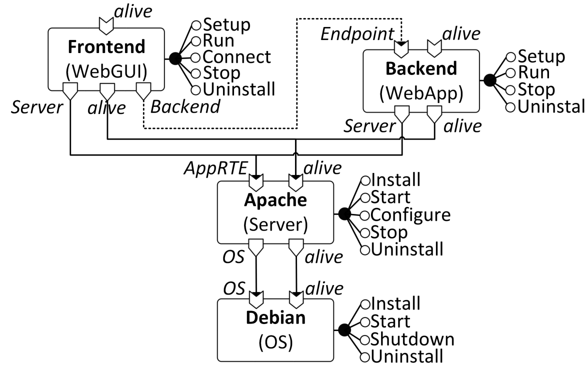


FIGURE 7.8: Motivating scenario: updated topology.

The updated topology permits to container nodes to witness whether they continue to be available (by providing the $\mathsf{alive}^{\mathsf{c}}$ capability), and to contained nodes to check whether their containers continue to be available (by assuming the $\mathsf{alive}^{\mathsf{r}}$ requirement). This requires to update the management protocol $\mathcal{M}_N = \langle \overline{s}_N, \rho_N, \chi_N, \tau_N, \varphi_N \rangle$ of each node $N \in T$ by substituting it with $\mathcal{M}'_N = \langle \overline{s}_N, \rho'_N, \chi'_N, \tau'_N, \varphi'_N \rangle$, which is built as follows:

- All states in S_N (but the initial one) can be reached by N only if the container of N continues to be available. Hence, the function ρ_N is

updated by making all states (but the initial one) assuming the requirement $\mathsf{alive}^{\mathsf{r}}_N$, in addition to the requirements they already assume. Formally:

$$\rho'_N(s) = \begin{cases} \rho_N(s) \cup \{\mathsf{alive}^{\mathsf{r}}_N\} & \text{if } s \neq \overline{s}_N \text{ and container}(N) \neq \bot \\ \rho_N(s) & \text{otherwise} \end{cases}$$

- Whenever N is not in its initial state, it can be considered as alive (as it is ensured that it has performed some operation to get there). To witness this fact, the function χ_N is updated by making all states (but the initial one) providing the $\mathsf{alive}^{\mathsf{c}}_N$ capability in addition to those they already provide. Formally:

$$\chi'_N(s) = \begin{cases} \chi_N(s) \cup \{\mathsf{alive}^{\mathsf{c}}_N\} & \text{if } s \neq \overline{s}_N \\ \chi_N(s) & \text{otherwise} \end{cases}$$

- Each transition in τ_N requires the container of N (if any) to be alive, and this means that each transition in τ_N has to constrain its executability to the satisfaction of the $\mathsf{alive}^{\mathsf{r}}_N$ requirement. Additionally, since all transitions (but those whose source/target is the initial state) connect two states offering the $\mathsf{alive}^{\mathsf{c}}_N$ capability, they can also maintain such capability while being executed. Formally:

$$\tau'_N = \{\langle s_1, P', X', o, s_2\rangle \mid \langle s_1, P, X, o, s_2\rangle \in \tau_N\}$$

where

$$P' = \begin{cases} P \cup \{\mathsf{alive}^{\mathsf{r}}_N\} & \text{if container}(N) \neq \bot \\ P & \text{otherwise} \end{cases}$$

and

$$X' = \begin{cases} X \cup \{\mathsf{alive}^{\mathsf{c}}_N\} & \text{if } s_1, s_2 \neq \overline{s}_N \\ X & \text{otherwise} \end{cases}$$

- Finally, the fault handling relation φ_N has to be extended to handle the potential fault of the $\mathsf{alive}^{\mathsf{r}}_N$ requirement (if any). If such requirement stops being satisfied, this means that the node in which N is contained has been hard reset, which in turns means that also N has been hard reset. Hence, φ_N has to be extended by adding all transitions handling the fault of the $\mathsf{alive}^{\mathsf{r}}_N$ requirement by making N go back to its initial state $\overline{s}_N$. Formally:

$$\varphi'_N = \varphi_N \cup \{\langle s, H, \overline{s}_N\rangle \mid s \in S_N - \{\overline{s}_N\} \ \wedge \ \mathsf{alive}^{\mathsf{r}}_N \in H \subseteq \rho'_N(s)\}$$

Example 7.6. The fault-aware management protocols in our motivating scenario can be updated as shown in Fig. 7.9, which illustrates the updated protocols of *Apache* and *Backend*[10]. In each state (other than the initial one), *Apache* and *Backend*

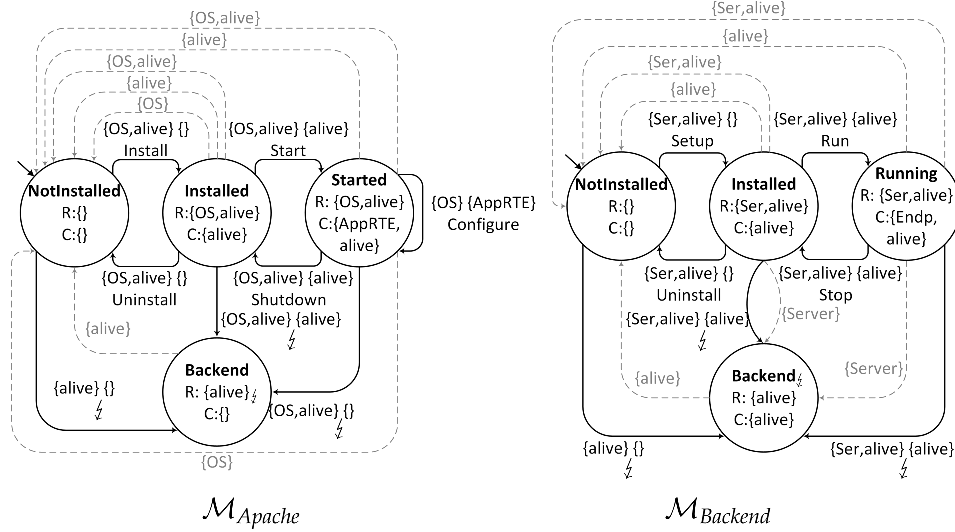


FIGURE 7.9: Example of management protocols with $\mathsf{alive}^{\mathsf{r}}$ requirements and $\mathsf{alive}^{\mathsf{c}}$ capabilities.

assume their $\mathsf{alive}^{\mathsf{r}}$ requirement, i.e. they assume their containers to continue to be available. They are also providing their $\mathsf{alive}^{\mathsf{c}}$ capability in such states, to witness to the nodes they contains (i.e., *Apache* contains *Frontend* and *Backend*, while *Backend* is not containing any node) that they continue to be there.

Notice that, whenever the $\mathsf{alive}^{\mathsf{r}}$ requirement of *Apache* or *Backend* is faulted, the corresponding node returns to its initial state. This models the fact that, whenever a container is uninstalled, the nodes it contains are uninstalled along with it. □

The above construction is recapped by the following definition.

Definition 7.11 (Enabling hard recovery). *Let $A = \langle T, b\rangle$ be a composite application, and let $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N\rangle$ with $\mathcal{M}_N = \langle \overline{s}_N, \rho_N, \chi_N, \tau_N, \varphi_N\rangle$. To* enable hard recovery*, A is adapted into a new composite application $A' = \langle T', b'\rangle$, which is built according to the following construction rules:*

$$T' = \{\langle S_N, R'_N, C'_N, O_N, \mathcal{M}'_N\rangle \mid \langle S_N, R_N, C_N, O_N, \mathcal{M}_N\rangle \in T\}$$

where

- $R'_N = \begin{cases} R_N \cup \{\mathsf{alive}^{\mathsf{r}}_N\} & \textit{if } container(N) \neq \bot \\ R_N & \textit{otherwise} \end{cases}$
- $C'_N = C_N \cup \{\mathsf{alive}^{\mathsf{c}}_N\}$*, and*
- $\mathcal{M}'_N = \langle \overline{s}_N, \rho'_N, \chi'_N, \tau'_N, \varphi'_N\rangle$*, with*

[10]We omit the updated protocols of *Frontend* and *Debian* since their update is similar to that of *Apache* and *Backend*.

– $\rho'_N(s) = \begin{cases} \rho_N(s) \cup \{\mathsf{alive}^{\mathsf{r}}_N\} & \textit{if } s \neq \overline{s}_N \textit{ and container}(N) \neq \bot \\ \rho_N(s) & \textit{otherwise} \end{cases}$

– $\chi'_N(s) = \begin{cases} \chi_N(s) \cup \{\mathsf{alive}^{\mathsf{c}}_N\} & \textit{if } s \neq \overline{s}_N \\ \chi_N(s) & \textit{otherwise} \end{cases}$

– $\tau'_N = \{\langle s_1, P', X', o, s_2\rangle \mid \langle s_1, P, X, o, s_2\rangle \in \tau_N\}$, *where*

  * $P' = \begin{cases} P \cup \{\mathsf{alive}^{\mathsf{r}}_N\} & \textit{if container}(N) \neq \bot \\ P & \textit{otherwise} \end{cases}$

  * $X' = \begin{cases} X \cup \{\mathsf{alive}^{\mathsf{c}}_N\} & \textit{if } s_1, s_2 \neq \overline{s}_N \\ X & \textit{otherwise} \end{cases}$

– $\varphi'_N = \varphi_N \cup \{\langle s, H, \overline{s}_N\rangle \mid s \in S_N - \{\overline{s}_N\} \ \wedge \ \mathsf{alive}^{\mathsf{r}}_N \in H \subseteq \rho'_N(s)\}$.

and

$$b'(r) = \begin{cases} \mathsf{alive}^{\mathsf{c}}_{N'} & \textit{if } r = \mathsf{alive}^{\mathsf{r}}_N \textit{ and container}(N) = N' \\ b(r) & \textit{otherwise} \end{cases}$$

7.5.2 Planning hard recovery

The construction rules enabling hard recovery (Def. 7.11), combined with the analyses presented in Sect. 7.3.2, permits analysing the management behaviour of a composite application by also considering the possibility of hard resetting a node N, to unlock the nodes contained in N and that are stuck in their sink states. For instance, the notion of validity (Def. 7.8) can be reused to check whether a hard recovery plan is valid, and it is also possible to analyse the effects of a hard recovery plan. More interestingly, we can automatically determine a plan recovering the desired global state of an application by simply visiting the graph associated with the labelled transition system modelling the application's management behaviour.

Example 7.7. Consider again our motivating scenario (Fig. 7.1), and suppose that the application is stuck in the global state reached in Fig. 7.7. By updating the modelling of the application as illustrated in Examples 7.5 and 7.6, it is possible to plan the (hard) recovery of the application from such "stuck" global state.

Essentially, *Backend* is stuck in *Backend*$_{\lightning}$, and the only way to get out of it is to remove its *alive* requirement, which in turn means to *Shutdown* and *Uninstall Apache* (to make it stop providing its *alive* capability). This results in resetting also the *Frontend*, which goes back to its initial state. Afterwards, we can re-*Install* and *Start* the *Apache* server, *Setup* and *Run* both the *Backend* and the *Frontend*, and *Connect* the *Frontend* (to the *Backend*).

The above listed operations build up the hard recovery plan in Fig. 7.10. It can be trivially verified that such plan is valid in the "stuck" global state reached in Fig. 7.7, and that it permits hard recovering the application by making all its nodes be up and running again. As we already mentioned, the above recovery plan can simply be determined with a visit of the graph associated with the transition

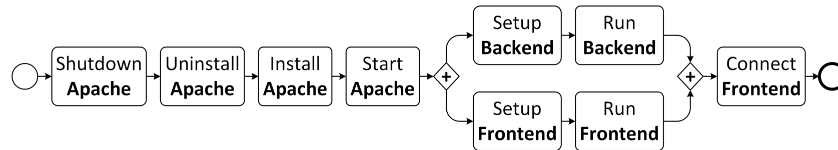


FIGURE 7.10: Example of hard recovery plan.

system defined by management behaviour of the application (whose modelling has been automatically updated according to the construction rules in Def. 7.11 — see Examples 7.5 and 7.6). □

7.6 Implementation

To illustrate the feasibility of our approach, we developed a new version[11] of BARREL (see Sect. 5.4), which permits editing and analysing fault-aware management protocols in TOSCA applications. The graphical user interface of BARREL 2.0 is written HTML5, while its back-end is written in TypeScript and JavaScript. In the following, we shall not deepen into implementation details, but rather we focus on how BARREL 2.0 can be used to edit and analyse existing TOSCA applications.

Setting the stage

The very first step is to import a valid CSAR package[12] (see Sect. 2.2) containing a TOSCA service template, along with all definitions of node types that are instantiated in its topology. Once the CSAR is loaded, the *Visualise*, *Edit*, and *Analyse* panes become selectable in the navigation bar (and the *Visualise* pane is selected by default).

Visualising applications

The *Visualise* pane graphically displays the application modelled in the imported CSAR (Fig. 7.11).

The name of the application is placed in the top-left corner of the *Visualise* pane. The application topology is visualised in the left-hand side of the pane, by drawing all nodes composing such topology, their requirements and capabilities (at the top and at the bottom of each node, respectively), and all relationships binding a requirements of a node with a capability of another node. Further information about each node (such as the node type

[11]The application can be accessed at http://di-unipi-socc.github.io/barrel/ with any modern web browser, like Microsoft Edge, Google Chrome or Mozilla Firefox. The source code is publicly available on GitHub at https://github.com/di-unipi-socc/barrel.

[12]CSAR packages can be imported by clicking on the *CSAR* option in the navigation bar of BARREL 2.0. A sample CSAR can be downloaded by clicking on *About*.

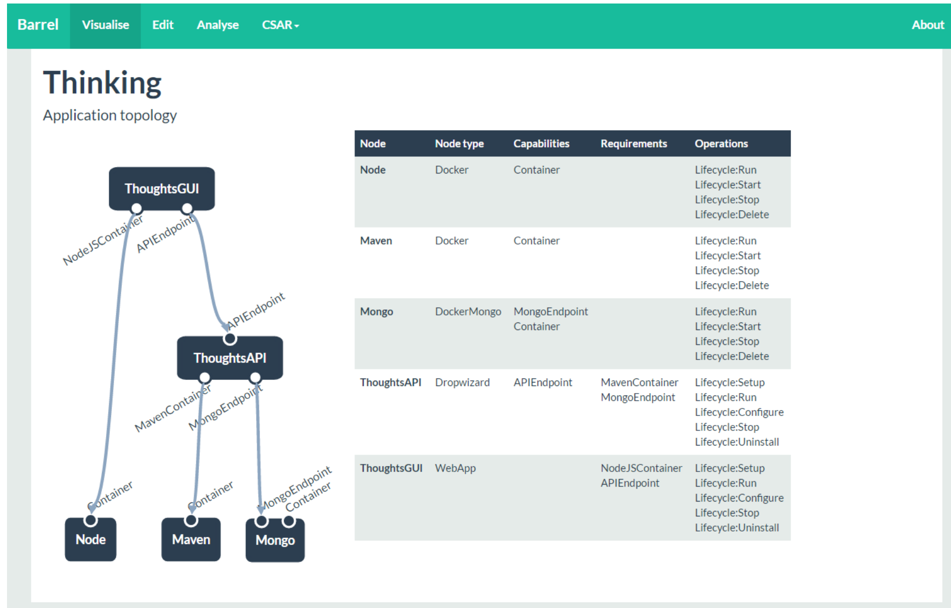


FIGURE 7.11: BARREL 2.0: *Visualise* pane.

or the management operations it offers) is listed in the table placed in the right-hand side of the *Visualise* pane.

Editing the fault-aware management protocols in an application

The *Edit* pane (Fig. 7.12) permits editing the fault-aware management protocols of the node types instantiated in the application topology.

The *Management protocol editor* permits selecting the node type to edit through a dedicated drop-down menu. Once a node type is selected, its fault-aware management protocol is displayed and can be modified by exploiting the tools right below it. The initial state can be selected through a dedicated drop-down menu, and the current values of ρ and χ (for all states) can be edited by clicking on the corresponding *Edit* button. Transitions can be added and removed from τ by clicking on the *Add* and *Remove* buttons, respectively. Similarly, fault-handling transitions can be added and removed from φ by clicking on the dedicated buttons. Finally, it is possible to automatically complete the displayed protocol: By clicking on the *Fault* button, all unhandled faults are default handled as discussed in Def. 7.5. By clicking on *Crashes*, instead, the protocol is updated by adding the crash operation $\lightning$ and all corresponding transitions (to permit analysing the management of the corresponding nodes, also if they are behaving unexpectedly — see Def. 7.10).

The updates applied to the management protocol of the currently selected node type can also be viewed in the XML source of such node type, by clicking on the *Show XML* button appearing in the top-right corner of the *Edit* pane. Once the fault-aware management protocols of all node types

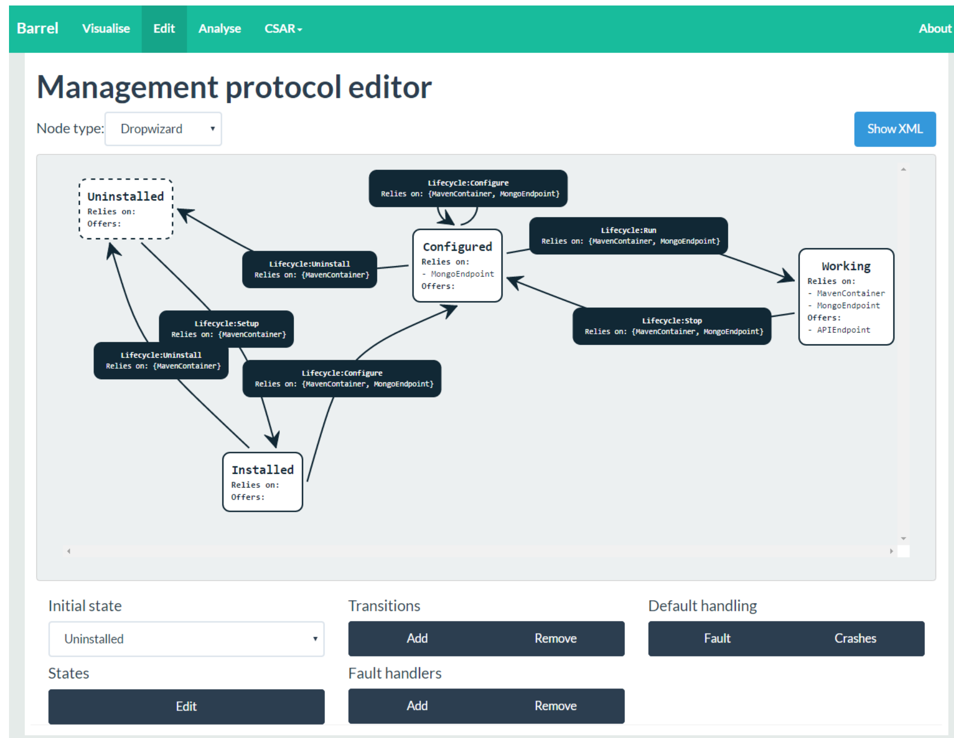


FIGURE 7.12: BARREL 2.0: *Edit* pane.

have been edited, the updated CSAR can be downloaded through the *CSAR Export* functionality (in the navigation bar).

Analysing the fault-aware management of an application

The *Analyse* pane (Fig. 7.13) permits interactively analysing the fault-aware management behaviour of the imported service template. It offers a *Simulator* for simulating the behaviour of the service template, and a *Planner* for automatically determining valid plans. Both simulation and planning can be carried out with hard recovery either enabled or disabled (by setting the corresponding option in the *Options* section).

The *Simulator* permits simulating sequences of operations and determining their effects on the whole application. More precisely, the table of the *Simulator* lists all the node templates in the application topology, each associated with its current state, the requirements it currently relies on, the capabilities it offers, and the operations actually available. Each operation is rendered as a green button if all the capabilities connected to requirements needed to execute it are currently available, otherwise it is rendered as a yellow (disabled) button. By clicking on an available operation, users can simulate its execution, and subsequently update the global state displayed by the simulator table. The update can result in faulting some requirements, which are then displayed as red buttons. If the fault of a requirement can be handled, the corresponding button is clickable, otherwise it is disabled.

Barrel Visualise Edit Analyse CSAR About

Options

Hard recovery: Off

Simulator

	State	Offered capabilities	Assumed requirements	Available operations
Node	Running	Container		Lifecycle:Stop
Maven	Running	Container		Lifecycle:Stop
Mongo	Stopped			Lifecycle:Start Lifecycle:Delete
ThoughtsAPI	Working	APIEndpoint	MavenContainer MongoEndpoint	Lifecycle:Stop
ThoughtsGUI	Running		NodeJSContainer	Lifecycle:Configure

Reset simulator

Planner

Starting global state

Node	State
Node	Unavailable
Maven	Unavailable
Mongo	Unavailable
ThoughtsAPI	Uninstalled
ThoughtsGUI	Uninstalled

The above **state is reachable** from the initial global state.

Switch states

Target global state

Node	State
Node	Running
Maven	Running
Mongo	Running
ThoughtsAPI	Working
ThoughtsGUI	Working

The above **state is reachable** from the initial global state.

The **target state** can be reached from the **starting state** as follows:

Operation Execute operation **Lifecycle:Run** on node **Node**
Operation Execute operation **Lifecycle:Run** on node **Maven**
Operation Execute operation **Lifecycle:Run** on node **Mongo**
Operation Execute operation **Lifecycle:Setup** on node **ThoughtsGUI**
Operation Execute operation **Lifecycle:Setup** on node **ThoughtsAPI**
Operation Execute operation **Lifecycle:Run** on node **ThoughtsGUI**

FIGURE 7.13: BARREL 2.0: *Analyse* pane.

By clicking on the button corresponding to a requirement, users can simulate the handling of such requirement, hence updating the global state displayed by the *Simulator*. An update can also result in permitting to hard recover a node that is stuck in its *Crashed* state. If this is the case, a red button *Hard recover* appears next to the node's state, and by clicking on such button the global state of the application is updated by resetting the node to its initial state. The simulation can be reset at any time, by clicking on *Reset simulator*.

With the *Simulator*, users can already perform many of the analyses described in this chapter. For instance, to check whether a plan is valid (see Def. 7.8), they just need to simulate its sequential traces and check that such traces can be executed in their entirety. To check whether a plan is fault-free (see Def. 7.9), they just need to check that no fault is generated by simulating any of its sequential traces. They can also compute the effects of a plan on states, capabilities and requirements by looking at the initial and final configurations displayed by the *Simulator* table.

The *Planner* automatically determines the sequence of operations and fault-handlers to be invoked to reach the *Target global state* from the *Starting global state*. Both global states can be set by associating each node with one of its states through a dedicated drop-down menu. To improve user experience, whenever the state of a node is changed, the plan is recomputed[13].

Concluding remarks

It is worth noting that BARREL 2.0 is already compatible with the OpenTOSCA open source ecosystem [15, 78]. BARREL 2.0 is indeed able to process CSARs developed with the visual editor Winery [78], and it produces CSARs that can be imported both in Winery [78] and in OpenTOSCA [15]. Of course, while both Winery [78] and OpenTOSCA [15] import the CSARs generated by BARREL 2.0, they do not properly process the information concerning fault-aware management protocols (since the extension to TOSCA we propose is not yet part of the TOSCA standard, and hence not yet supported in the OpenTOSCA open source environment).

[13] BARREL 2.0 enables this by maintaining a matrix whose (i, j)-th element is the next step on the shortest path from the reachable global state i to the reachable global state j, and by exploiting this matrix to reconstruct the shortest path from the starting global state to the target global state. The matrix, as well as the shortest path from a global state to another, are created and maintained by implementing the Floyd-Warshall algorithm [58].

7.7 Case study: *Thinking*

We hereby illustrate how fault-aware management protocols (as well as BARREL 2.0) can be fruitfully exploited to analyse and orchestrate the management of the *Thinking* application[14] (see Sect. 5.5).

7.7.1 Enabling fault handling and hard recovery

First, the modelling of *Thinking* is automatically extended to enable fault handling, and to permit hard recovering components that are stuck in their sink state. The latter requires to slightly modify the application topology as indicated by Def. 7.11. More precisely, the topology of *Thinking* (Fig. 5.9) is extended by adding an *alive* requirement to *ThoughtsGui* and *Thoughts-Api*, by adding an *alive* capability to all nodes, and by adding two relationships connecting the *alive* requirement of *ThoughtsGui* to the *alive* capability of *Node* and the *alive* requirement of *ThoughtsApi* to the *alive* capability of *Maven*. The resulting application topology is depicted in Fig. 7.14.

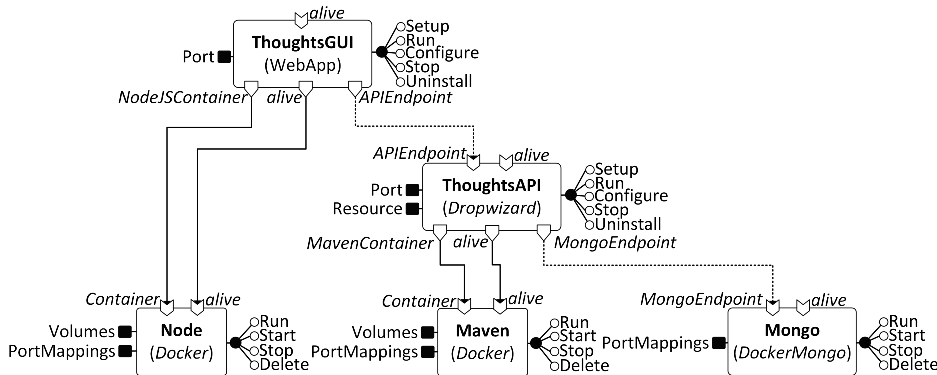


FIGURE 7.14: Adaptation of *Thinking*'s topology enabling hard recovery.

The management protocols of all nodes in *Thinking* are also automatically extended, by implementing the construction rules given by Defs. 7.5, 7.10, and 7.11. Fig. 7.15 shows the fault-aware management protocols (a) for *Mongo* and (b) for *Maven* and *Node*. Such protocols extend the management protocols displayed in Figs. 5.10 and 5.11, by including the additional "sink" states $\mathit{DockerMongo}_{\lightning}$ and $\mathit{Docker}_{\lightning}$ (Def. 7.5), by including the $\lightning$ transitions that permit analysing what would happen to the other nodes in *Thinking* if *Mongo*, *Maven*, or *Node* behave unexpectedly (Def. 7.10), and by providing the *alive* capability in all their states but the initial one (Def. 7.11). Notice that there is no fault handling transition, as there is no requirement assumption that can be faulted in any state.

Fig. 7.16 illustrates the fault-aware management protocol of *Thoughts-Api*. It extends the original management protocol of *ThoughtsApi* (Fig. 7.16)

[14]The case study has been run on an Ubuntu 16.04 LTS virtual machine, with 32 GB of storage and 8 GB of memory.

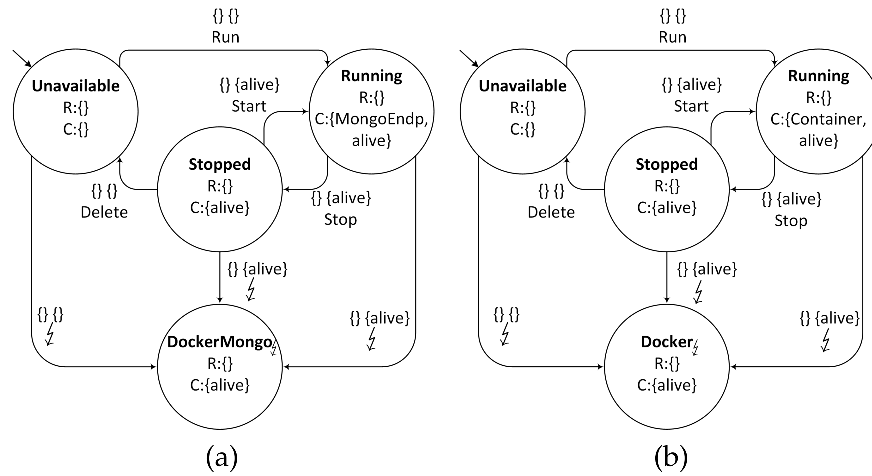


FIGURE 7.15: Fault-aware management protocols for the nodes (a) of type *Docker-Mongo* and (b) of type *Docker*.

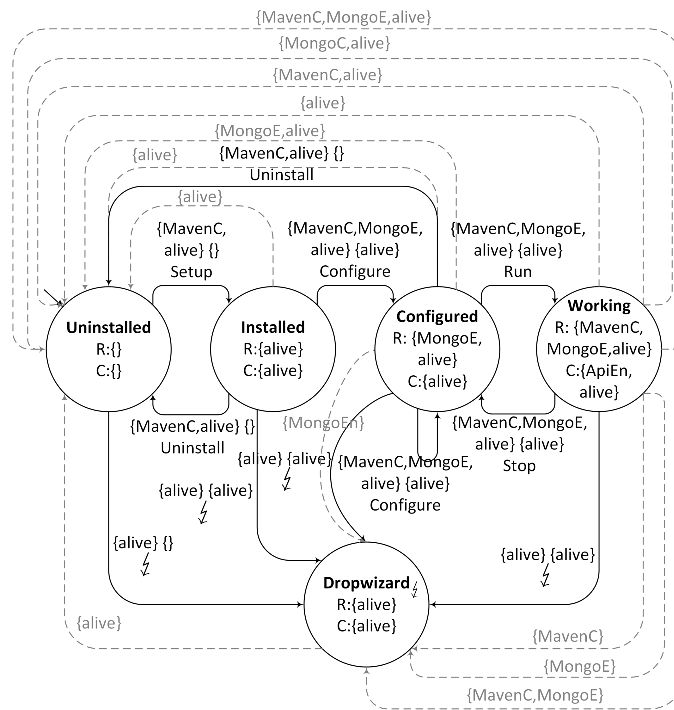


FIGURE 7.16: Fault-aware management protocol for *ThoughtsApi*'s node type.

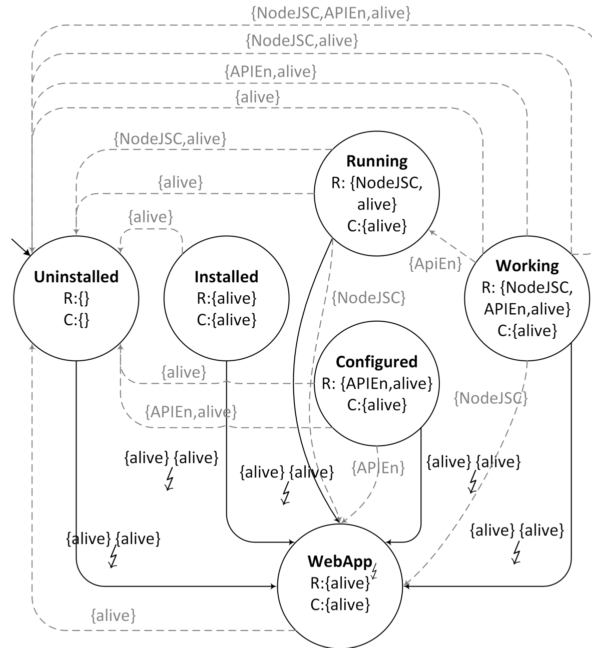


FIGURE 7.17: Fault-aware management protocol for *ThoughtsGui*'s node type.

by including an additional state *Dropwizard*$_{↯}$ acting as a "sink" for all unhandled faults of requirements (Def. 7.5), by including the ↯ transitions that permit analysing the worst-possible effects on *Thinking* of a misbehaving *ThoughtsApi* (Def. 7.10), and by handling the fault of the *alive* requirement by making the *ThoughtsApi* go back to its initial state (Def. 7.11).

Finally, Fig. 7.17 shows[15] the fault-aware management protocol of *ThoughtsGui*, which extends the management protocol in Fig. 5.13. We add the state *ThoughtsGui*$_{↯}$, the ↯ transitions leading to such state from any other state, and the fault handling transitions. The latter are similar to those displayed in Fig. 7.16: All transitions handling faults including the *alive* requirement target the initial state *Uninstalled*, the transition handling the fault of the solely *APIEndpoint* requirement in the *Working* state targets the state *Configured*, and all other fault-handling transitions target the "sink" state ThoughtsGui$_{↯}$.

7.7.2 Planning the undeployment of *Thinking*'s GUI and API

Consider a running instance of *Thinking*, which can be obtained by executing the valid deployment plan illustrated in Sect. 5.5.2. Suppose also that we wish to undeploy its components *ThoughtsGui* and *ThoughtsApi* (i.e., that we wish to come back to the global state where *Mongo* is the only component installed and running).

[15]For reasons of readability, we omit to display the transitions corresponding to the execution of available management operations. They can be found in Fig. 5.13.

The problem of finding a plan that starts from a global state and reaches another global state can be solved with a breadth-first search of the graph associated with the transition system of an application's management behaviour (see Sect. 5.3). By applying this approach to our situation, we discover that one of the shortest, valid sequences of operations that permit undeploying the components components *ThoughtsGui* and *ThoughtsApi* from a running instance of *Thinking* is that displayed in Fig. 7.18. If we execute such sequence of operations, we effectively come back to the situation where no container but that of *Mongo* is installed and running (Fig. 7.19).

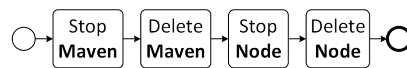


FIGURE 7.18: An undeployment plan for *Thinking*.

```
jacopo@yellow:~$ docker ps -a
CONTAINER ID  IMAGE  COMMAND                    CREATED         STATUS        PORTS
fc5646d8b38e  mongo  "/entrypoint.sh mongo"  8 minutes ago  Up 8 minutes  27017/tcp
jacopo@yellow:~$
```

FIGURE 7.19: Snapshot displaying the effective undeployment of the components *ThoughtsGui* and *ThoughtsApi* from a running instance of *Thinking*.

It is worth noting that the plan discovered in this section is much simpler than that discovered in Sect. 5.5.3. This is because, with fault-aware management protocols, a node can remove one of its capabilities even if other nodes are relying on it, since such nodes can handle the corresponding fault with a dedicated transition. The same is not true for the analyses based on "plain" management protocols (Chapter 5), where nodes cannot remove their capabilities while other nodes are actually relying on them.

7.7.3 Analysing the effects of a misbehaving component

Consider another instance of *Thinking* (such as that in Fig. 7.20), and suppose that its *ThoughtsApi* component unexpectedly[16] crashes. Which are the effects on the rest of the application?

As we discussed in Sect. 7.4, we can determine the worst-possible effects on the other components of *Thinking* by invoking the "crash" operation $\lightning$ of *ThoughtsApi*, and by looking at the subsequent evolution of the global state. Such evolution is displayed in Fig. 7.21. Notice that the reached global state effectively models reality: After the crash of *ThoughtsApi*, both*ThoughtsGui* and *Mongo* are still running, but *ThoughtsGui* is not capable to effectively serve its client because it cannot connect to *ThoughtsApi*. Indeed, if we try to connect to *ThoughtsGui*, no thought is displayed in it (since the GET request sent to *ThoughtsApi* does not receive any answer — Fig. 7.22).

[16]To simulate the unexpected crash of *ThoughtsApi* we actually killed the corresponding process in the *Maven* container.

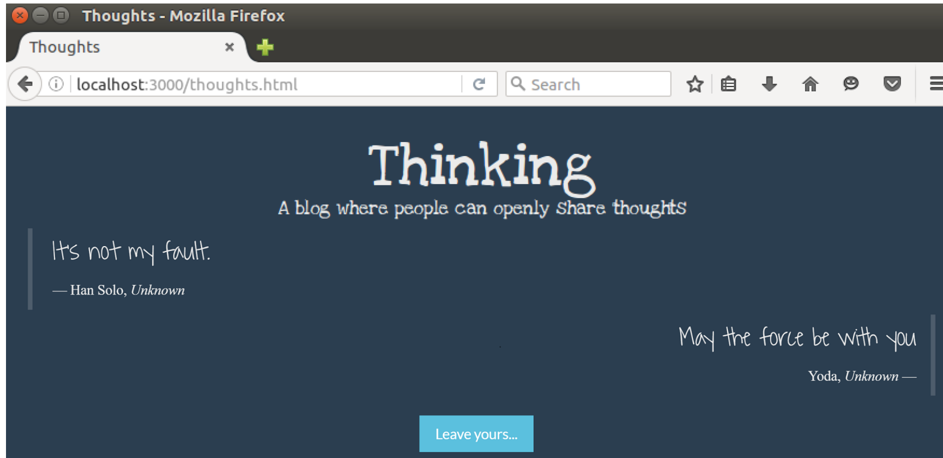


FIGURE 7.20: A running instance of *Thinking*.

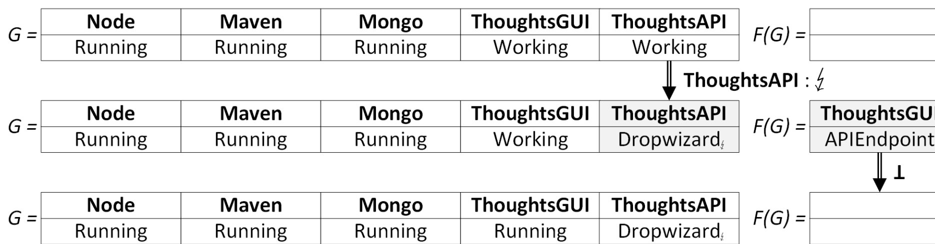


FIGURE 7.21: Evolution of the global state of *Thinking* after injecting the "crash" of *ThoughtsApi*.

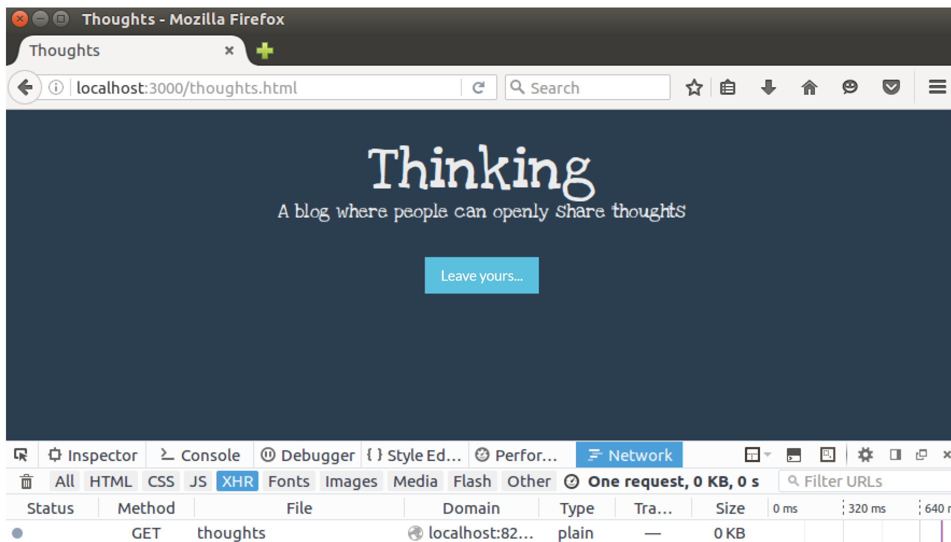


FIGURE 7.22: Snapshot displaying the effects on *ThoughtsGui* of a crashing *ThoughtsApi*.

To automatically recover the considered instance of *Thinking*, we need to determine a hard recovery plan (since *ThoughtsApi* is stuck in its sink state $\text{Dropwizard}_{\lightning}$). Please recall that the modelling of *Thinking* has already been extended to support hard recovery (Figs. 7.14, 7.15, 7.16, and 7.17). This means that the hard recovery plan can be determined by simply

performing a breadth-first search of the graph associated with the labelled transition system modelling the management behaviour of *Thinking*. With this approach, we discover that one of the shortest, valid sequences of operations that permit hard recovering a running instance of *Thinking* is that displayed in Fig. 7.23 (which also illustrates the corresponding evolution of the global state of the considered instance of *Thinking*). By executing such sequence of operations, we effectively recover our instance of *Thinking* (as illustrated by Fig. 7.24).

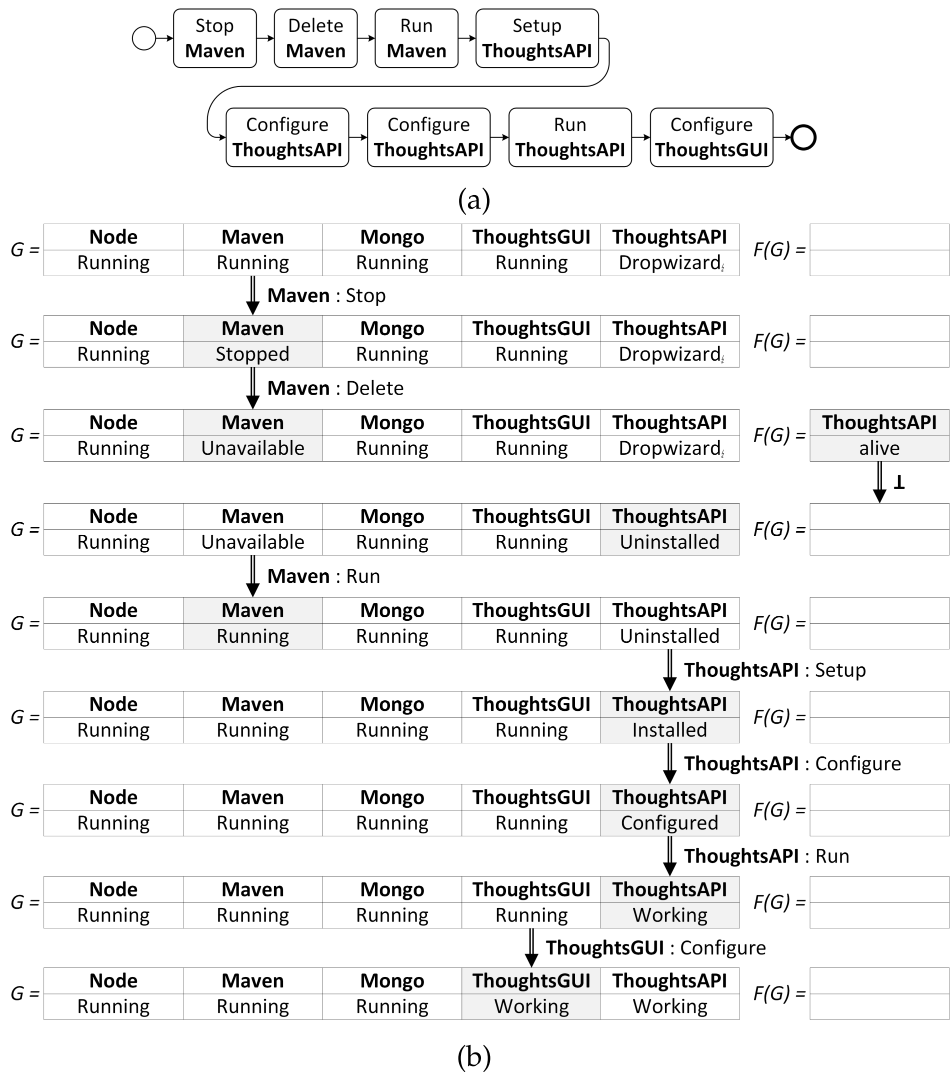


FIGURE 7.23: (a) Hard recovery plan, and (b) corresponding evolution of the global state of *Thinking*.

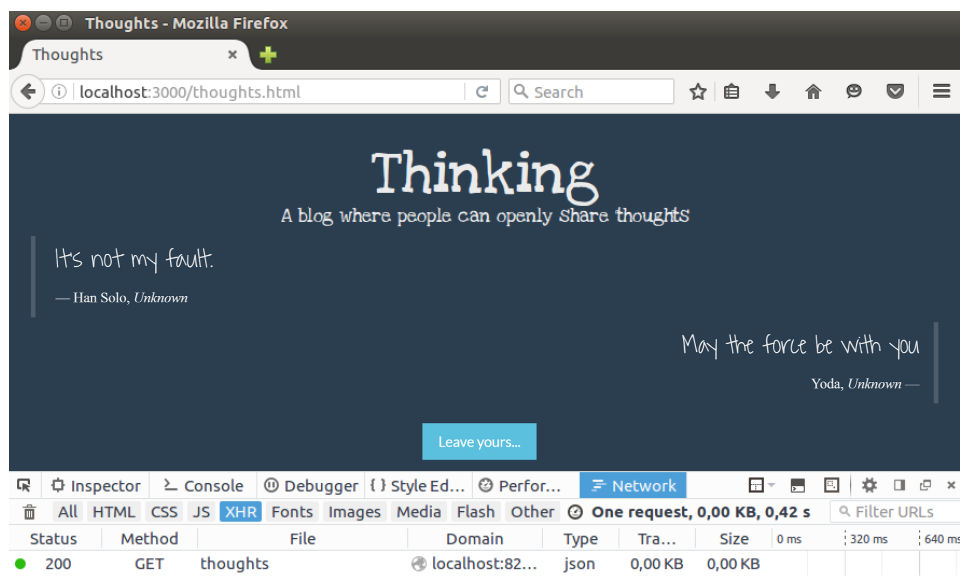


FIGURE 7.24: Snapshot displaying the effective recovery of *Thinking*.

Chapter 8

Related work

We hereafter discuss related work, by separately treating existing solutions for syntactically matching cloud applications (Sect. 8.1), for modelling their management behaviour (Sect. 8.2), and for matching cloud applications by taking into account their management behaviour (Sect. 8.3).

8.1 Syntactic matching of cloud applications

Our work started from the observation that while the matching between service templates and node types is indicated in the TOSCA primer [95] as a way to instantiate abstract node types, no formal definition of matching is given either in the TOSCA specification [94] or in the TOSCA primer. A first concrete definition of matching for TOSCA has been proposed by Binz et al. [14] to define a way to merge TOSCA applications by matching entire portions of their topologies. The definition of matching of single service components employed by Binz et al. is however very strict, as two application components are considered to match only if they expose the same qualified name. We aim at contributing to the TOSCA specification by formalising less strict notions of syntactic matching between service templates and node types, and by illustrating how to adapt matched (fragments of) service templates to reuse them to concretely implement desired node types.

In the following we will position our approaches for syntactically matching and adapting cloud applications (presented in Chapters 3 and 4) with respect to other solutions for the matching and adapting available services or cloud applications.

Matching and adapting (Web) services

The problem of how to match (Web) services has been extensively studied in recent years. Many approaches are ontology-aware [99], like for instance the ontology-aware matchmaker for OWL-S services described by Klusch et al. [75]. Other approaches are behaviour-aware, like the trace-based matching of YAWL services defined by Brogi and Popescu [31], the

behavioural congruence for OWL-S services defined by Bonchi et al. [22], or the graph transformation based matching defined by Corrales et al. [44] and the heuristic black-box matching for WS-BPEL processes described by Eshuis and Grefen [54]. The main difference between the aforementioned approaches and ours is the type of information considered when matching single nodes. For instance, the matching levels considered by Klusch et al., as well as those considered by Eshuis and Grefen, are all defined in terms of input and output data, while we consider also technology requirements and capabilities, properties and policies.

On the other hand, many proposals of QoS-aware service matching are available, like for instance the approaches by Mahdikhani et al. [84] or by Mokhtar et al. [87]. Generally speaking, our approach for syntactically matching cloud applications differs from most QoS-aware matching approaches since it compares *types* rather than actual *values* of features.

A type-based definition of matching has been defined by Eliassen and Mehus [53] to type check "stream flows" for interactive distributed multimedia applications. While the context of stream flows is different from ours, two of the matching conditions considered by Eliassen and Mehus resemble our notions of exact and plug-in matching, even if for simpler service abstractions.

Summing up, to the best of our knowledge, our definition of matching is the first definition of (TOSCA) node matching that takes into account both functional and non-functional features, by relying both on types and on ontologies to overcome non-relevant syntactic differences.

It is also worth highlighting that our notions of *plug-in*, *renaming-based* and *white-box* matching share the basic objectives with alternating refinement relations [3] (and more in general with the notion of simulation [107]). Indeed, they all check whether an available component is capable of offering all the features/options of a desired component (without imposing additional requirements). However, while alternating refinement and simulation rely on both the signature and behaviour of components, our notions of matching only rely on the signature of components, as this is what can be described in TOSCA.

It is hence interesting to extend TOSCA by permitting to describe the behaviour of a component, and to extend our notions of matching to take into account such behaviour information. Such extensions have been presented in Chapters 5 and 6, respectively.

Matching and adapting cloud applications

The development of systematic approaches to adapt and reuse existing software is widely recognised as one of the crucial problems in software engineering [59, 115]. In spite of the increasing availability of cloud solutions,

currently platform-specific code often needs to be manually modified to reuse existing solutions in cloud-based applications. This is obviously an expensive and error-prone activity, as pointed out by Tran et al. [110], both for the learning curve and for the testing phases needed.

Various efforts have been recently oriented to try devising systematic approaches to reuse cloud applications. For instance, Di Martino et al. [48] and Hamdaqa et al. [68] propose two solutions to transform platform-agnostic source code of applications into platform-specific applications, provided that they developed according to model-driven methodologies. In contrast, our approach is not restricted to applications developed with a specific methodology, nor it requires the availability of the source code of applications, and it is hence applicable also to third-party services.

Guillén et al. [66] propose a framework which allows developers to develop cloud-based services as if they were "in-house" applications. Cloud deployment information must be provided in a separate file, and a middleware layer employs source and deployment information to generate the artifacts to be deployed on cloud platforms. We believe that our approach improves that of Guillén et al. in two ways. First, with the approach by Guillén et al., the reuse of an application requires invoking the middleware layer, while with our approach the adaptation is performed only once. Second, Guillén et al. always require to write source code, while our approach only requires to edit the application specification.

In general, most existing approaches to the reuse of cloud services support a from-scratch development of cloud-agnostic applications, and do not account for the possibility of adapting existing (third-party) cloud-based services. In contrast, our approach proposes a way to adapt existing cloud applications, by relying on TOSCA as the standard for cloud interoperability, and to support an easy reuse of third-party services.

Other approaches worth mentioning are those proposed by Arnold et al. [7, 8], by Andrikopoulos et al. [4], by Hirmer et al. [69], and by Di Cosmo et al. [46]. Arnold et al. [7, 8] provide a method to deploy and provision SOA solution, in which patterns are used to structure and constraint composite applications, without binding to specific resources, and without specifying provisioning actions. Andrikopoulos et al. [4] propose a way to detect the optimal deployment for a cloud-based solution, by employing a highly abstracted topology. Nevertheless, these approaches require an application developer to structure the topology of the IT solution to provision, and based on this structure the provisioning can be detected. In contrast, our approach gives developers all the freedom in choosing whether to structure the whole topology, or to abstract all the needed components as a single (and standalone) module. The latter solution obviously eases the development of cloud applications, thus flatting the learning curve needed to

provision them. Furthermore, both Arnold et al. and Andrikopoulos et al. focus only on the provisioning of cloud application, while our approach gives developers the means to also describe the configuration operations needed to manage their applications.

Hirmer et al. [69] permit developers to describe only the components that are specific of a composite application, whose topology is then automatically completed based upon the components' requirements. However, Hirmer et al. do not permit developers to specify what they expect from the nodes and relationships inserted to complete a topology. In contrast, our approach permits them to abstractly describe any component in an application topology (as well as the features it has to provide). Furthermore, our approach implements such components by reusing (fragments of) existing applications, while Hirmer et al. build the completion from scratch (by inserting single nodes and relationships). Thus, while a solution obtained with our approach is expected to be already configured, a solution returned with the approach proposed by Hirmer et al. may require to be manually configured.

Di Cosmo et al. [46] present a toolchain that permits automating the assembly and deployment of composite cloud applications. The toolchain takes as input a description of the components forming an application, of their requirements, and of the available cloud resources, and it automatically determines an optimal deployment plan for such application. The deployment plan assigns application components to cloud resources by minimising resource consumption, it specifies how to interconnect components to satisfy their requirements, and it describes how to configure each resource to effectively run the components to be deployed on it (e.g., which software packages to install in a virtual machine). Hence, the approach by Di Cosmo et al. differs from our approach mainly because of the targeted problem: Di Cosmo et al. focus on determining the optimal deployment of the components forming a composite application, while we focus on allowing developers to abstractly describe the components forming their applications and on reusing (fragments of) other applications to concretely implement such components.

Finally, it is worth noting that the novelty of the proposed approaches does not reside in the type of adaptation techniques applied to service templates. Indeed, our method exploits well-known existing patterns of adaptation (e.g., Gamma et al. [62], Becker et al. [10]) to adapt TOSCA templates. The novelty of our approaches is rather that, in contrast with traditional adaptation approaches (e.g., Bracciali et al. [23], Kongdenfha et al. [76]), no additional code must be developed to reuse existing cloud-based services. We exploit the possibilities provided by TOSCA of mapping exposed features

onto internal ones, and of entirely delegating the management of such mappings to TOSCA containers [95].

8.2 Modelling the behaviour of cloud applications

The problem of automating application management is well-known in computer science. After the cloud revolution, it has become even more prominent because of the complexity of both applications and platforms [36]. This is witnessed by the proliferation of so-called "configuration management systems", such as Chef [39] or Puppet [104]. These management systems provide domain-specific languages to model the desired configuration for a software solution, and employ a client-server model to ensure that such configuration is met. However, the lack of a machine-readable representation of how to effectively manage application components inhibits the possibility of performing automated analyses on components' configurations and dependencies.

A first attempt to model and automate the management of composite cloud applications was the Aeolus component model [47]. Aelous shares the same underlying idea of (fault-aware) management protocols: Developers describe the behaviour of their components through finite-state machines, which are then composed to model the management behaviour of composite applications, and to automate their management. Engage [57] and Juju [73] are two other approaches for processing application descriptions to automatically deploy composite applications. Fault-aware management protocols however differ from Aelous, Engage, and Juju because they permit explicitly modelling faults and injecting failures in application components, analysing the effects of faults, and reacting to faults to restore a desired application state.

The rigorous engineering of fault-tolerant systems is a well-known problem in computer science [35], with many existing approaches targeting the design and analysis of such systems. For instance, Johnsen et al. [72] propose a way to design object-oriented systems by starting from fault-free systems, and by subsequently refining such design by handling different types of faults. Qiang et al. [105] and Betin Can et al. [12] instead focus on fault-localisation, with the final objective of permitting to redesign a system to avoid the occurrence of such a fault. These approaches differ from ours because they aim at obtaining applications that "never fail", since all potential faults have to be identified at design/development time and properly handled. Our approach is instead more recovery-oriented [37], since we focus on applications where faults possibly occur, and we permit designing applications capable of being recovered.

Similar considerations apply to Grunske et al. [65], Kaiser et al. [74], and Alhosban et al. [2], which however share with our approach the basic idea of modelling faults in single components and of composing the obtained models according to the dependencies between such components (i.e., according to the application topology).

Friedrich et al. [60] proposes an approach to handle faults in service-based processes which is very close in the spirit to ours. As we do for composite applications, service-based processes are described with a model-based approach, and the description of a process includes the possible repair actions for each of its activities. This permits checking recoverability of actions at design time, and generating recovery plans whenever a fault is detected (by an external monitoring tool). The approach by Friedrich et al. however differs from our approach mainly because of the application domain, which permits them exploiting different techniques (such as heuristics based on branching probabilities) to carry out their analyses, and since they assume faults to happen one at a time. Their approach also differs from ours because they do not cope with services whose actual behaviour is different from that modelled in the process.

Durán and Salaün [52] propose a decentralised approach to deploy and reconfigure cloud applications in presence of failures. They model a composite applications as a set of interconnected virtual machines, each equipped with a configurator managing its instantiation and destruction. The deployment and reconfiguration of the whole application is then orchestrated by a manager interacting with virtual machine configurators. The approach by Durán and Salaün shares with our approach the objective of providing a decentralised and fault-aware management of a composite application, by specifying the management of each component separately. However, it differs from our approach since it permits specifying inter-component dependencies, but it is not possible to describe whether they are "horizontal" (i.e., a component requires another to be up and running) or "vertical" dependencies (i.e., a component is installed/deployed on another). Additionally, it focuses on recovering virtual machines that have been terminated because of environmental faults, while we also permit describing how components react to application-specific faults.

Liggesmeyer and Rothfelder [83] propose an approach to identify failures in a system whose components' behaviour is described by finite state machines. Even though the analyses are quite different, the modelling proposed by Liggesmeyer and Rothfelder is quite similar to ours. It indeed relies on a sort of requirements and capabilities to model the interaction among components, and it permits "implicitly" modelling how components behave in presence of single/multiple faults. Our modelling is a strict generalisation of that by Liggesmeyer and Rothfelder, since the state of a

component can change not only because of requirement unsatisfaction but also because of invoked operations, and since it permits "explicitly" handling faults (viz., fault handling transitions are distinct from those modelling the normal behaviour of a component). Similar considerations apply to the approach proposed by Chen et al. [40], whose modelling is also based on finite state machines with input and output channels (which permit fault communication and propagation by components).

UFIT [67] is a tool for verifying fault-tolerance of systems. It permits modelling the behaviour of systems with timed automata, some of whose transitions explicitly represent how the system reacts to the occurrence of faults. Even if it models fault transitions in a way similar to ours, UFIT differs from our approach since it targets standalone systems and does not provide any mechanism to easily compose the automata modelling the behaviour of multiple systems.

Other approaches worth mentioning are those by de Lemos and Fiadeiro [81], and by Nagatou and Watanabe [90]. The way in which our approach models fault-awareness by relying on the interactions between components, as well as the idea of analysing/recovering faults through sequences of atomic transactions (until a desired state is reached), are indeed inspired by de Lemos and Fiadeiro. Instead, the idea of relying on fault injection to determine the effects of unpredictable faults is inspired by Nagatou and Watanabe.

In summary, to the best of our knowledge, the approach we proposed in this thesis is the first that permits automatically orchestrating the management of composite applications under the assumption that faults possibly occur during such management, thus requiring to explicitly model how an application reacts to their occurrence. It does so by following the common idea of modelling each component separately, and of deriving the management behaviour of a composite application by properly combining the behaviour of its components.

Finally, it is worth highlighting that we have investigated the possibility of employing composition-oriented automata (like *interface automata* [1]) to model valid plans directly as the language accepted by the automaton obtained by composing the automata modelling the management protocols of the components of an application. The main drawbacks of such an approach are the size of the obtained automaton (which grows exponentially with the number of application components and hence makes the automaton scarcely readable even for simple applications), and the need of recomputing the automaton whenever a new component is added or its management protocol is modified.

8.3 Behaviour-aware matching of cloud applications

The problem of how to match existing software components (by also taking into account their behaviour) has been extensively studied in recent years. Cavallaro et al. [38] propose an approach to automatically replace services based upon their functional interface and behaviour models. Cavallaro et al. [38] focus on many-to-many mappings among service operations, since such mappings have to hold in whatever state of the service. Our approach is more flexible in the sense that it is capable of mapping a single operation to different operation sequences depending on the state in which such operation is invoked. Furthermore, Cavallaro et al. [38] generate an adapter script that has to be passed to a proxy any time the corresponding operation is invoked. Our approach instead adapts the service once for all (by translating the function f into a set of TOSCA plans).

Reussner et al. [106] describe how to adapt components based on parametric contracts, which permit modifying their interfaces depending on context properties in a way potentially more expressive than ours. Likewise us, Reussner et al. [106] exploit finite state machines to model interaction protocols. However, the solution by Reussner et al. [106] differs from our approach since it does not make explicit the relation between context properties and protocols, thus loosing information on what it is concretely reachable in a composite environment.

Closely related approaches are also those by Motahari Nezhad et al. [88], Inverardi and Tivoli [70], and Bennaceur and Issarny [11], even if they propose approaches to synthesize mediators among service and service clients, while we focus on matching. Our work shares with Motahari Nezhad et al. [88] (and previous papers by the same authors) the idea of exploiting, at the same time, functional interfaces and behaviour models to match an operation of a service with multiple operations of another service. Our function f (Def. 6.7) is indeed pretty similar to the *interface mapping* proposed by Motahari Nezhad et al. [88]. However, our approach is fully automated while that employed by Motahari Nezhad et al. [88] is semi-automated.

The approach by Inverardi and Tivoli [70] shares its baselines with that by Motahari Nezhad et al. [88], and synthesises adapters in the form of interaction protocols. Bennaceur and Issarny [11] instead exploit ontologies and constraint programming to infer one-to-one, one-to-many, and many-to-many correspondences between interfaces of components, and synthesise adapters in the form of labelled transition systems. Our approach synthesises adapters that are considerably simpler than those produced by Inverardi and Tivoli [70] or by Bennaceur and Issarny [11], as the adapters we synthesise just consist in the determined f functions (Def. 6.7). Furthermore, our notion of f-simulation extends the notion of one-to-many operation mapping introduced by Bennaceur and Issarny [11].

Summing up, to the best of our knowledge, ours is the first fully automated approach for reusing TOSCA application components, which takes into account both functional and extra-functional features of TOSCA application components, and which relies on the widely accepted idea of exploiting behaviour models to match operations, and on behaviour simulation [107] to go beyond non-relevant operation mismatches.

Last, but not least, it is worth discussing how our notion of (protocol) simulation relates with existing notions. *Stuttering* [34] and *branching* [45, 63] bisimulations are the closest to our f-simulation. Stuttering bisimulation [34] is defined on Kripke structures [79], where labels appear on states but not on transitions, and therefore there is no need for a function for translating transition labels. Branching bisimulation [45, 63] is instead defined on labelled transition systems, and permits matching a transition only with another transition having the same label, followed by a sequence of silent steps. Hence, it does not requires any translation function which behaves as f, i.e. which maps a (desired) labelled transition to sequences of (available) labelled transitions. Furthermore, in both stuttering [34] and branching [45, 63] bisimulations, the intermediate states occurring in a sequence of transitions should satisfy some equivalence constraints, while in f-simulation these states are completely hidden behind the definition of the transition system $\Rightarrow$ (Def. 6.5). This abstraction is safe in our context, since the interactions among components relies just on requirements and capabilities, which are annotated in the transitions and checked by the definition of f-simulation (see Remark 6.1). Moreover, since we consider a preorder rather than an equivalence, the additional transitions of intermediate states do not play any role.

Somehow, our notion of f-simulation is closer in the spirit to *stuttering* as intended by Lamport [80]. We indeed consider a phenomenon recurring in various aspects of computer science (e.g., compilers), namely the simulation of a high-level operation by a sequence of lower-level steps.

Chapter 9

Conclusions

How to deploy and flexibly manage complex composite applications over heterogeneous cloud platforms is a serious challenge in enterprise IT [82]. This thesis aims to contribute solving this challenge by focussing on two major issues, namely (I_1) automating the deployment and management of composite cloud applications, and (I_2) supporting a vendor-agnostic design of composite cloud applications.

In this chapter we summarise the research contributions contained in this thesis (see Sect. 9.1), and we discuss how such contributions can contribute solving issues I_1 and I_2 (see Sect. 9.2). We also give some perspectives for future work (see Sect. 9.3).

9.1 Summary of contributions

To (I_1) automate the deployment and management of composite cloud applications, by also (I_2) supporting their vendor-agnostic design, this thesis aims at advancing the state-of-the-art for (o_1) modelling and (o_2) analysing composite cloud applications, and for (o_3) reusing them (see Sect. 1.1).

We now summarise the research contributions presented in this thesis. The summary follows the workflow in Fig. 1.1, which distinguishes the contributions concerning o_1 and o_2 from those concerning o_3, and which shows which chapters directly exploit the results presented in another chapter.

Modelling and analysing composite cloud applications

By relying on topology graphs [20] for describing the structure of composite cloud applications, and on TOSCA [94] as the reference language for representing them, we have proposed a compositional modelling that permits specifying the management behaviour of an application's components, by also taking into account the fact that faults eventually occur while managing complex applications [42, 43].

In Chapter 5 we have illustrated how the management behaviour of an application component can be modelled by *management protocols*, specified as finite state machines whose states and transitions are associated with

conditions defining the consistency of the states of a component and constraining the executability of its management operations. Such conditions are defined on the requirements of a component, and each requirement of a component has to be fulfilled by a capability of another component. We have also illustrated how to compose the protocols of the components forming a composite cloud application to derive its management behaviour, and how this permits automating various analyses concerning its management (e.g., determining whether management plans are valid, which are their effects, or which plans permit reaching certain application configurations).

In Chapter 7 we have extended management protocols to permit modelling and analysing composite cloud applications in presence of faults. More precisely, we have proposed *fault-aware management protocols*, which permit modelling how components behave when faults occur. We have then illustrated how to compose the protocols of the components forming a composite cloud application to derive its fault-aware management behaviour, and how to analyse and automate the management of composite cloud applications in a fault-resilient manner. Furthermore, after noticing that the actual behaviour of application components may differ from their described behaviour (e.g., because of non-deterministic bugs [64]), we have shown how the unexpected behaviour of components can be naturally modelled by automatically completing their fault-aware management protocols, and how this permits analysing the effects of a misbehaving component on the rest of a cloud application (in the worst case). We have also proposed a way to hard recover composite cloud applications that are stuck because a fault was not properly handled, or because of a misbehaving component.

Fostering the reuse of composite cloud applications

To foster the reuse of composite cloud applications, in Chapter 3 we have formalised four notions of syntactic matching between an available application and a desired component. We have also explained how a matched application can be adapted to exactly match the target component, and hence be reused to concretely implement it. To illustrate the feasibility of the proposed approaches, we have implemented and tested them on a plastic repository of validated TOSCA applications.

The aforementioned notions of matching permit reusing an application only in its entirety. Hence, to concretely implement a desired component one could end up deploying unnecessary software (i.e., portions of the reused application that are not needed to concretely implement the desired application component). To tackle this issue, in Chapter 4 we have further extended our approach for reusing composite cloud applications by introducing TOSCAMART, a method that permits reusing only the fragment of

an application topology that is actually necessary for implementing a desired application component.

Finally, in Chapter 6, we have extended the aforementioned notions of syntactic matching to include the behaviour information in management protocols. More precisely, we have defined a notion of simulation [107] between management protocols, and we have exploited such notion to extend the conditions constraining exact and plug-in matching. We have then relaxed the notion of simulation into that of f-simulation (which permits simulating a desired operation with a a sequence of available operations), and we have exploited f-simulation to further relax plug-in matching. We have also described a coinductive [71] procedure to compute the function f determining an f-simulation between two management protocols, which has been proved to be sound and complete.

9.2 Assessment of contributions

As we discussed in Sect. 1.1, this thesis aims at advancing the state-of-the-art for (o_1) modelling and (o_2) analysing composite cloud applications, and for (o_3) reusing them. We hereby assess the research contributions in this thesis first with respect to o_1 and o_2, and then with respect to o_3.

Modelling and analysing composite cloud applications

Fault-aware management protocols can play a foundational role for modelling and analysing the management of composite cloud applications. Indeed, to the best of our knowledge, they constitute the first compositional approach that permits modelling and analysing the stateful management behaviour of the components forming an application, by also taking into account that faults possibly occur while managing complex composite applications over heterogeneous clouds (see Sect. 8.2).

The feasibility of approaches based on fault-aware management protocols has been illustrated with the BARREL 2.0 prototype (see Sect. 7.6), while their potential has been highlighted by exploiting them to automate the deployment and management of the *Thinking* case study (see Sects. 5.5 and 7.7). The *Thinking* case study has highlighted that the modelling and analysis techniques based on fault-aware management protocols can be fruitfully exploited not only at design time (to validate management plans, and to determine their effects), but also at run time (to automatically determine the management plans that permit reaching a desired application configuration, or which restore such configuration after the actual configuration has changed — e.g., because of faults or misbehaving components).

On the other hand, a full-fledged approach for modelling and analysing composite cloud applications requires to solve also other problems that

have not been tackled in this thesis. Below we discuss three of them, namely faults that might be generated during management protocols transitions, applications whose topology is dynamic, and including cost and QoS in the modelling of applications.

Faults generated during transitions. It is important to mention that (as per Defs. 7.6 and 7.7) we have focussed on the consistency of states, by considering as faults only the requirements that are assumed in a global state and whose corresponding capability is not provided in the same global state.

However, there might be cases where a requirement is assumed by a component, and another component performs a transition during which the capability satisfying such requirement is not maintained (even if it is available in the source and target states). It would be worthy investigating whether this might generate problems in real-world scenarios, and (if this is the case) how to properly adapt the composition rules defining the fault-aware management behaviour of a composite application.

Dynamic topologies. Fault-aware management protocols can be easily adapted to cope with applications whose topology is dynamic. Indeed, to deal with applications whose components may dynamically (dis)appear, such components should be added to the application topology, and the binding function relating requirements and capabilities should be updated accordingly. This would be useful, for instance, to cope with the horizontal scaling of an application's components, as it would permit adding or removing replicas of a component to the application topology whenever such component has to be scaled out or scaled in.

Adapting our modelling and analysis techniques to cope with dynamic topologies would also be beneficial for exploiting them to manage other kinds of applications, which are characterised by a high churn of nodes (e.g., microservices-based applications, or fog applications).

Modelling cost and QoS. Fault-aware management protocols do not take into account costs nor QoS, since the focus of the thesis is on automatically coordinating the management of the components forming a composite cloud application (by also taking into account that faults potentially occur while managing complex applications). Cost and QoS are however important factors for cloud applications [6], and fault-aware management protocols should hence be extended to take into account also such factors. For instance, we should permit modelling how much does it cost (in terms of money or time) to reside in a certain state or to perform a certain operation. This would permit devising analysis techniques to determine the cost (in terms of money or time) to maintain/drive an application in/to a given configuration, or to determine the cheapest or fastest management plans that permit changing the actual configuration of an application.

Fostering the reuse of composite cloud applications

The matching techniques presented in this thesis can constitute a fruitful support to foster the reuse of composite cloud applications, and to speed-up their design and development. Developers can indeed describe only the application components that are specific to their solutions (e.g., those they implemented), along with abstract descriptions of the management infrastructures such components need to run. Such abstract descriptions could then be concretely implemented by matching, adapting, and reusing (fragments of) already existing solutions.

The feasibility and potential of our notions of matching have been tested by running a proof-of-concept implementation of the syntactic matching over a plastic repository of TOSCA applications. The effectiveness has also been discussed by formally proving termination and soundness of TOSCA-MART, and by assessing the behaviour-aware matching (by formally proving termination, soundness, and completeness of the coinductive algorithm determining a f-simulation between two protocols).

It is worth noting that, in general, most existing approaches to the reuse of cloud applications support a development from-scratch of applications, and do not account for the possibility of adapting existing third-party applications. To the best of our knowledge, the syntactic matching approaches presented in Chapters 3 and 4 advance the state-of-the-art as they are the first approaches that permit reusing (fragments of) existing cloud applications, by relying on TOSCA as the reference standard for cloud interoperability, and to support an easy reuse of third-party services (see Sect. 8.1). The behaviour-aware matching discussed in Chapter 6 is also advancing the state-of-the-art on the reuse of composite cloud applications. Indeed, it constitutes the first approach for reusing composite applications that takes into account both functional and extra-functional features of their components, and which relies on the widely accepted idea of exploiting behaviour models to match operations and on behaviour simulation to abstract from non-relevant operation mismatches (see Sect. 8.3).

On the other hand, in order to select (fragments of) composite cloud applications that can be effectively reused to implement abstractly specified components, some problems have still to be investigated and addressed. Below we discuss three of them, namely the full integration of the proposed techniques, the assessment of the substitutability assumption made by TOSCA, and the inclusion of cost and QoS in our matching approaches.

Full integration of the proposed techniques. The behaviour-aware matching proposed in Chapter 6 permits reusing applications only in their entirety. To permit reusing only the fragments of such applications that are actually necessary to implement a desired component, the current implementation

of TOSCAMART should be integrated with the behaviour-aware matching that we presented in Chapter 6.

Furthermore, the behaviour-aware matching proposed in Chapter 6 does not take into account faults, since the notions of simulation are defined on "plain" management protocols (as per Chapter 5). The notions of simulation and of behaviour-aware matching should be extended to permit comparing the fault-aware management protocol of (a fragment of) an available application with that of a desired component.

Substitutability assumption. All the notions of matching we presented in this thesis are based on the substitutability assumption made by TOSCA, which states that a component can be made concrete by substituting it with a composite application, provided that the latter exposes the same features as the former on its boundaries [95]. The truthfulness of such an assumption should be tested on repositories of real-word TOSCA applications, which unfortunately are not available at the moment.

Cost-aware and QoS-aware matching. All notions of matching contained in this thesis do not take into account costs nor QoS. This is because the focus of the thesis is on enacting the reuse of composite cloud applications by matching their syntactic signature and their management behaviour with respect to those of a desired component.

Cost and QoS are however important factors for cloud applications [6]. The notions of matching should hence be extended to take into account also the cost or QoS of composite cloud applications, as this would permit selecting, among the matched applications, those leading to lower costs or to better time performances, for instance.

9.3 Possible directions for future work

Fault-aware management protocols can play a foundational role for modelling and analysing the management of composite cloud applications. The feasibility of approaches based on fault-aware management protocols has been illustrated with BARREL (see Sect. 7.6), while their potential has been shown by exploiting them to automate the deployment and management of the *Thinking* case study (see Sects. 5.5 and 7.7).

The next step is the development of an orchestrator for composite cloud applications based upon fault-aware management protocols. The orchestrator should input a composite application and its desired configuration, and it should ensure that such application configuration is reached and maintained. The orchestrator should indeed exploit management protocols to determine the managemen plan leading the application to the desired configuration (from the initial situation where no application component is installed), and it should execute such management plan. Then, whenever

a fault occurs and changes the actual application configuration (or whenever the desired configuration is changed), the orchestrator should automatically determine a management plan to restore the desired application configuration. The development of such orchestrator is left for future work.

The reuse techniques presented in this thesis can also contribute to support a vendor-agnostic design of composite cloud applications, and to automate the deployment and management of composite applications. The feasibility and potential of our notions of matching have been tested, and their effectiveness has been formally assessed.

On the other hand, a full-fledged support for modelling, analysing, and reusing composite cloud applications requires also solutions to problems that have not been tackled in this thesis. Some of them were listed in Sect. 9.2, and the corresponding directions for future work are listed below.

Faults generated during transitions. There might be cases where a requirement is assumed by a component, and another component performs a transition during which the capability satisfying such requirement is not maintained (even if it is available in the starting and target states). We plan to investigate whether this might generate problems in real-world scenarios, and (if so) to properly adapt the composition rules defining the fault-aware management behaviour of a composite application.

Dynamic topologies. To deal with applications whose components may dynamically (dis)appear, it should be enough to add such components to the application topology, and to update the binding function relating requirements and capabilities. A formalisation of this is in the scope of our immediate future work.

Modelling cost and QoS. The modelling and analyses techniques based upon fault-aware management protocols do not take into account costs and QoS. The extension of such techniques to include cost and QoS properties is in the scope of our future work.

Full integration of the proposed techniques. Our behaviour-aware matching approach (see Chapter 6) does not take into account faults, and it permits reusing applications only in their entirety. We plan to extend our behaviour-aware matching approach to permit comparing the fault-aware management protocols (hence including faults in the comparison), and to integrate it with TOSCAMART (to permit reusing only the fragments of application that are actually necessary for implementing a desired component).

Substitutability assumption. The truthfulness of the substitutability assumption made by TOSCA (which states that a component can be made concrete by substituting it with a composite application, provided that the latter exposes the same features as the former on its boundaries [95]) should

be tested on repositories of real-word TOSCA applications, which unfortunately are not available at the moment. The assessment of such an assumption is hence left for future work.

Cost-aware and QoS-aware matching. All notions of matching contained in this thesis do not take into account costs or QoS. The extension of such notions to include QoS and costs in the selection of to-be-reused components are in the scope of our future work.

Bibliography

[1] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *Proceedings of the 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of Software Engineering*. In ESEC/FSE-9. ACM, 2001, pages 109–120. ISBN: 1-58113-390-1. DOI: 10.1145/503209.503226.

[2] Amal Alhosban, Khayyam Hashmi, Zaki Malik, Brahim Medjahed, and Salima Benbernou. Bottom-up fault management in service-based systems. *ACM transactions on Internet technologies*, 15(2):7:1–7:40, 2015. ISSN: 1533-5399. DOI: 10.1145/2739045.

[3] Rajeev Alur, Thomas A. Henzinger, Orna Kupferman, and Moshe Y. Vardi. Alternating refinement relations. In *Proceedings of the 9th international conference on Concurrency Theory*. In CONCUR '98. Springer, 1998, pages 163–178. DOI: 10.1007/BFb0055622.

[4] Vasilios Andrikopoulos, Santiago Gómez Sáez, Frank Leymann, and Johannes Wettinger. Optimal distribution of applications in the cloud. In *Advanced Information Systems Engineering: 26th international conference, CAiSE 2014, Thessaloniki, Greece, June 16-20, 2014. Proceedings*. Matthias Jarke, John Mylopoulos, Christoph Quix, Colette Rolland, Yannis Manolopoulos, Haralambos Mouratidis, and Jennifer Horkoff, editors. Springer International Publishing, 2014, pages 75–90. DOI: 10.1007/978-3-319-07881-6_6.

[5] Vasilios Andrikopoulos, Anja Reuter, Santiago Gómez Sáez, and Frank Leymann. *A GENTL approach for cloud application topologies*. In. *Service-Oriented and Cloud Computing: 3rd european conference, ESOCC 2014, Manchester, UK, September 2-4, 2014. Proceedings*. Massimo Villari, Wolf Zimmermann, and Kung-Kiu Lau, editors. Springer Berlin Heidelberg, 2014, pages 148–159. DOI: 10.1007/978-3-662-44879-3_11.

[6] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010. ISSN: 0001-0782. DOI: 10.1145/1721654.1721672.

[7] William Arnold, Tamar Eilam, Michael H. Kalantar, Alexander V. Konstantinou, and Alexander Totok. Automatic realization of SOA deployment patterns in distributed environments. In *Service-Oriented Computing – ICSOC 2008: 6th international conference, Sydney, Australia, December 1-5, 2008. Proceedings*. Athman Bouguettaya, Ingolf Krüger, and Tiziana Margaria, editors. Volume 5364. In Lecture Notes in Computer Science. Springer, 2008, pages 162–179. ISBN: 978-3-540-89647-0. DOI: 10.1007/978-3-540-89652-4_15.

[8] William Arnold, Tamar Eilam, Michael H. Kalantar, Alexander V. Konstantinou, and Alexander Totok. Pattern based SOA deployment. In *Service-Oriented Computing — ICSOC 2007: 5th international conference, Vienna, Austria, September 17-20, 2007. Proceedings*. Bernd J. Krämer, Kwei-Jay Lin, and Priya Narasimhan, editors. Volume 4749. In Lecture Notes in Computer Science. Springer, 2007. ISBN: 978-3-540-74973-8. DOI: 10.1007/978-3-540-74974-5_1.

[9] Paolo Baldan, Andrea Corradini, Hartmut Ehrig, and Reiko Heckel. Compositional semantics for open Petri nets based on deterministic processes. *Mathematical Structures in Computer Science*, 15(01):1–35, 2005. DOI: 10.1017/S0960129504004311.

[10] Steffen Becker, Antonio Brogi, Ian Gorton, Sven Overhage, Alexander Romanovsky, and Massimo Tivoli. Towards an Engineering Approach to Component Adaptation. In Ralf H. Reussner, Judith A. Stafford, and Clemens A. Szyperski, editors, *Architecting Systems with Trustworthy Components*. Volume 3938, in Lecture Notes in Computer Science, pages 193–215. Springer, 2006. ISBN: 978-3-540-35800-8. DOI: 10.1007/11786160_11.

[11] Amel Bennaceur and Valérie Issarny. Automated synthesis of mediators to support component interoperability. *IEEE Transactions on Software Engineering*, 41(3):221–240, 2015. DOI: 10.1109/TSE.2014.2364844.

[12] Aysu Betin Can, Tevfik Bultan, Mikael Lindvall, Benjamin Lux, and Stefan Topp. Eliminating synchronization faults in air traffic control software via design for verification with concurrency controllers. *Automated Software Engineering*, 14(2):129–178, 2007. DOI: 10.1007/s10515-007-0008-2.

[13] Tobias Binz. Personal communication. November 22nd, 2013.

[14] Tobias Binz, Uwe Breitenbücher, Oliver Kopp, Frank Leymann, and Andreas Weiss. Improve resource-sharing through functionality- preserving merge of cloud application topologies. In *Proceedings of the 3rd international conference on Cloud Computing and Service Science,*

CLOSER 2013, 8-10 May 2013, Aachen, Germany. F. Desprez, D. Ferguson, E. Hadar, F. Leymann, M. Jarke, and M. Helfert, editors. SciTePress, 2013, pages 96–103. DOI: 10.5220/0004378000960103.

[15] Tobias Binz, Uwe Breitenbücher, Florian Haupt, Oliver Kopp, Frank Leymann, Alexander Nowak, and Sebastian Wagner. *OpenTOSCA – a runtime for TOSCA-based cloud applications*. In. *Service-Oriented Computing: 11th international conference, ICSOC 2013, Berlin, Germany, December 2-5, 2013, Proceedings*. Samik Basu, Cesare Pautasso, Liang Zhang, and Xiang Fu, editors. Springer, Berlin, Heidelberg, 2013, pages 692–695. DOI: 10.1007/978-3-642-45005-1_62.

[16] Tobias Binz, Uwe Breitenbücher, Oliver Kopp, and Frank Leymann. Automated Discovery and Maintenance of Enterprise Topology Graphs. In *Proceedings of the 6th IEEE international conference on Service-Oriented Computing and Applications, SOCA 2013*. IEEE Computer Society, 2013, pages 126–134. DOI: 10.1109/SOCA.2013.29.

[17] Tobias Binz, Uwe Breitenbücher, Oliver Kopp, and Frank Leymann. Migration of enterprise applications to the cloud. *Information Technology*, 56(3):106–111, 2014. ISSN: 1611-2776. DOI: 10.1515/itit-2013-1032.

[18] Tobias Binz, Uwe Breitenbücher, Oliver Kopp, and Frank Leymann. *TOSCA: Portable automated deployment and management of cloud applications*. In. *Advanced Web Services*. Athman Bouguettaya, Z. Quan Sheng, and Florian Daniel, editors. Springer, New York, NY, 2014, pages 527–549. ISBN: 978-1-4614-7535-4. DOI: 10.1007/978-1-4614-7535-4_22.

[19] Tobias Binz, Gerd Breiter, Frank Leymann, and Thomas Spatzier. Portable Cloud Services Using TOSCA. *IEEE Internet Computing Magazine*, 16(03):80–85, 2012. DOI: 10.1109/MIC.2012.43.

[20] Tobias Binz, Christoph Fehling, Frank Leymann, Alexander Nowak, and David Schumm. Formalizing the Cloud through Enterprise Topology Graphs. In *Cloud Computing (CLOUD), 2012 IEEE 5th international conference on*. IEEE Computer Society, 2012, pages 742–749. DOI: 10.1109/CLOUD.2012.143.

[21] Filippo Bonchi, Antonio Brogi, Andrea Canciani, and Jacopo Soldani. Behaviour-aware matching of cloud applications. In *Proceedings of the 10th international symposium on Theoretical Aspects of Software Engineering, TASE 2016*. IEEE, 2016, pages 117–124. DOI: 10.1109/TASE.2016.32.

[22] Filippo Bonchi, Antonio Brogi, Sara Corfini, and Fabio Gadducci. A Net-based approach to web services publication and replaceability. *Fundamenta Informaticae*, 94(3-4):305–330, 2009. DOI: 10.1145/944217.944233.

[23] Andrea Bracciali, Antonio Brogi, and Carlos Canal. A formal approach to component adaptation. *Journal of Sys. and Software*, 74(1):45–54, 2005. ISSN: 0164-1212. DOI: 10.1016/j.jss.2003.05.007.

[24] Uwe Breitenbücher, Tobias Binz, Oliver Kopp, and Frank Leymann. Vinothek - A Self-Service Portal for TOSCA. English. In *Proceedings of the 6th central european workshop on services and their composition (ZEUS'14)*. Volume 1140. In CEUR Workshop Proceedings. CEUR-WS.org, 2014, pages 69–72.

[25] Antonio Brogi, Andrea Canciani, and Jacopo Soldani. Fault-aware application management protocols. In *Service-Oriented and Cloud Computing: 5th IFIP WG 2.14 european conference, ESOCC 2016, Vienna, Austria, September 5-7, 2016, Proceedings*. Marco Aiello, Broch Einar Johnsen, Schahram Dustdar, and Ilche Georgievski, editors. Springer, 2016, pages 219–234. ISBN: 978-3-319-44482-6. DOI: 10.1007/978-3-319-44482-6_14.

[26] Antonio Brogi, Andrea Canciani, and Jacopo Soldani. Modelling and analysing cloud application management. In *Service-Oriented and Cloud Computing: 4th european conference, ESOCC 2015, Taormina, Italy, September 15-17, 2015, Proceedings*. Schahram Dustdar, Frank Leymann, and Massimo Villari, editors. Springer, 2015, pages 19–33. DOI: 10.1007/978-3-319-24072-5_2.

[27] Antonio Brogi, Andrea Canciani, and Jacopo Soldani. Modelling the behaviour of management operations in TOSCA. Tech. Rep., University of Pisa. 2015.

[28] Antonio Brogi, Andrea Canciani, Jacopo Soldani, and Pengwei Wang. A Petri net-based approach to model and analyze the management of cloud applications. In *Transactions on Petri Nets and Other Models of Concurrency XI*. Maciej Koutny, Jörg Desel, and Jetty Kleijn, editors. In LNCS Transactions on Petri Nets and Other Models of Concurrency. Springer Berlin Heidelberg, 2016, pages 28–48. ISBN: 978-3-662-53401-4. DOI: 10.1007/978-3-662-53401-4_2.

[29] Antonio Brogi, Andrea Canciani, Jacopo Soldani, and PengWei Wang. Modelling the behaviour of management operations in cloud-based applications. In *Proceedings of the international workshop on Petri Nets and Software Engineering, PNSE 2015, Brussels, Belgium, June 22-23,*

2015. Daniel Moldt, Heiko Rölke, and Harald Störrle, editors. Volume 1372. In CEUR Workshop Proceedings. CEUR-WS.org, 2015, pages 191–205.

[30] Antonio Brogi and Sara Corfini. Behaviour-aware discovery of web service compositions. *International Journal on Web Service Research*, 4(3):1–25, 2007. DOI: 10.4018/jwsr.2007070101.

[31] Antonio Brogi and Razvan Popescu. Service adaptation through trace inspection. *International Journal of Business Process Integration and Management*, 2(1):9–16, 2007. DOI: 10.1504/IJBPIM.2007.014100.

[32] Antonio Brogi and Jacopo Soldani. Finding available services in TOSCA-compliant clouds. *Science of computer programming*, 115–116:177–198, 2016. ISSN: 0167-6423. DOI: 10.1016/j.scico.2015.09.004.

[33] Antonio Brogi, Jacopo Soldani, and PengWei Wang. TOSCA in a nutshell: promises and perspectives. In *Service-Oriented and Cloud Computing: 3rd european conference, ESOCC 2014, Manchester, UK, september 2-4, 2014. Proceedings*. Massimo Villari, Wolf Zimmermann, and Kung-Kiu Lau, editors. Springer, 2014, pages 171–186. ISBN: 978-3-662-44879-3. DOI: 10.1007/978-3-662-44879-3_13.

[34] Michael C. Browne, Edmund M. Clarke, and Orna Grümberg. Characterizing finite Kripke structures in propositional temporal logic. *Theoretical Computer Science*, 59(1):115 –131, 1988. ISSN: 0304-3975. DOI: 10.1016/0304-3975(88)90098-9.

[35] Michael Butler, Cliff Jones, Alexander Romanovsky, and Elena Troubitsyna. *Rigorous development of complex fault-tolerant systems*. Volume 4157 of *Lecture Notes in Computer Science*. Springer-Verlag Berlin Heidelberg, 2006. ISBN: 3540482652. DOI: 10.1007/11916246.

[36] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. Cloud computing and emerging IT platforms: vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems*, 25(6):599 –616, 2009. DOI: 10.1016/j.future.2008.12.001.

[37] George Candea, Aaron B. Brown, Armando Fox, and David Patterson. Recovery-oriented computing: building multitier dependability. *IEEE Computer*, 37(11):60–67, 2004. DOI: 10.1109/MC.2004.219.

[38] Luca Cavallaro, Elisabetta Di Nitto, and Matteo Pradella. An automatic approach to enable replacement of conversational services. In

Service-Oriented Computing: 7th international joint conference, ICSOC-ServiceWave 2009, Stockholm, Sweden, November 24-27, 2009. Proceedings. Springer-Verlag, 2009, pages 159–174. ISBN: 978-3-642-10382-7. DOI: 10.1007/978-3-642-10383-4_11.

[39] Chef. Opscode. URL: https://www.opscode.com/chef.

[40] Lu Chen, Jian Jiao, and Jiping Fan. Fault propagation formal modeling based on stateflow. In *Proceedings of the 1st international conference on Reliability Systems Engineering, icrse 2015*. IEEE, 2015, pages 1–7. DOI: 10.1109/ICRSE.2015.7366480.

[41] Allan Cheng, Javier Esparza, and Jens Palsberg. Complexity results for 1-safe nets. In *Foundations of software technology and theoretical computer science*. Springer, 1993, pages 326–337. DOI: 10.1016/0304-3975(94)00231-7.

[42] Antonio Cisternino. Personal communication. October 15th, 2015.

[43] Richard I. Cook. How complex systems fail. Cognitive Technologies Laboratory, University of Chicago. URL: http://web.mit.edu/2.75/resources/random/HowComplexSystemsFail.pdf. 1998.

[44] Juan Carlos Corrales, Daniela Grigori, and Mokrane Bouzeghoub. BPEL processes matchmaking for service discovery. In *On the Move to Meaningful Internet Systems 2006: OTM confederated international conferences CoopIS, DOA, GADA, and ODBASE, Montpellier, France, October 29 - November 3, 2006. Proceedings*. In ODBASE'06/OTM'06. Springer-Verlag, 2006, pages 237–254. ISBN: 3-540-48287-3, 978-3-540-48287-1. DOI: 10.1007/11914853_15.

[45] Rocco De Nicola and Frits Vaandrager. Three logics for branching bisimulation. *Journal of the ACM*, 42(2):458–487, 1995. ISSN: 0004-5411. DOI: 10.1145/201019.201032.

[46] Roberto Di Cosmo, Michael Lienhardt, Ralf Treinen, Stefano Zacchiroli, Jakub Zwolakowski, Antoine Eiche, and Alexis Agahi. Automated synthesis and deployment of cloud applications. In *Proceedings of the 29th acm/ieee international conference on automated software engineering*. In ASE '14. ACM, 2014, pages 211–222. ISBN: 978-1-4503-3013-8. DOI: 10.1145/2642937.2642980.

[47] Roberto Di Cosmo, Jacopo Mauro, Stefano Zacchiroli, and Gianluigi Zavattaro. Aeolus: A component model for the cloud. *Information and Computation*, 239(0):100 –121, 2014. DOI: 10.1016/j.ic.2014.11.002.

[48] Beniamino Di Martino, Dana Petcu, Roberto Cossu, Pedro Goncalves, Tamas Mahr, and Miguel Loichate. Building a mOSAIC of clouds. In, *Euro-Par 2010 parallel processing workshops*. Volume 6586, in Lecture Notes in Computer Science, pages 571–578. Springer, 2011. ISBN: 978-3-642-21877-4. DOI: 10.1007/978-3-642-21878-1_70.

[49] Distributed Management Task Force, Inc. Cloud Infrastructure Management Interface (CIMI). URL: http://www.dmtf.org/sites/default/files/standards/documents/DSP0264_1.0.0.pdf. 2013.

[50] Distributed Management Task Force, Inc. Open Virtualization Format (OVF) Specification. URL: http://www.dmtf.org/sites/default/files/standards/documents/DSP0243_2.1.0.pdf. 2014.

[51] Docker. The Docker user guide. URL: https://docs.docker.com/userguide/. 2015.

[52] Francisco Durán and Gwen Salaün. Robust and reliable reconfiguration of cloud applications. *Journal of Systems and Software*, 122:524–537, 2016. ISSN: 0164-1212. DOI: 10.1016/j.jss.2015.09.020.

[53] Frank Eliassen and S. Mehus. Type checking stream flow endpoints. In *MIDDLEWARE'98: IFIP international conference on Distributed Systems Platforms and Open Distributed Processing*. Springer-Verlag, 1998, pages 305–320. ISBN: 1-85233-088-0. DOI: 10.1007/978-1-4471-1283-9_19.

[54] Rik Eshuis and Paul Grefen. Structural matching of bpel processes. In *Proceedings of the 5th IEEE european conference on Web Services, ecows 2007*. IEEE Computer Society, 2007, pages 171–180. ISBN: 0-7695-3044-3. DOI: 10.1109/ECOWS.2007.26.

[55] Christoph Fehling, Frank Leymann, Ralph Retter, Walter Schupeck, and Peter Arbitter. *Cloud computing patterns: Fundamentals to design, build, and manage cloud applications*. Springer, 2014. DOI: 10.1007/978-3-7091-1568-8.

[56] Nicolas Ferry, Alessandro Rossini, Franck Chauvel, Brice Morin, and Arnor Solberg. Towards model-driven provisioning, deployment, monitoring, and adaptation of multi-cloud systems. In *CLOUD 2013: 6th IEEE International Conference on Cloud Computing*. Lisa O'Conner, editor. IEEE Computer Society, 2013, pages 887–894. DOI: 10.1109/CLOUD.2013.133.

[57] Jeffrey Fischer, Rupak Majumdar, and Shahram Esmaeilsabzali. Engage: a deployment management system. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*. In PLDI '12. ACM, 2012, pages 263–274. ISBN: 978-1-4503-1205-9. DOI: 10.1145/2254064.2254096.

[58] Robert W. Floyd. Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962. DOI: 10.1145/367766.368168.

[59] William B. Frakes and Kyo Kang. Software reuse research: status and future. *IEEE Transactions on Software Engineering*, 31(7):529–536, 2005. DOI: 10.1109/TSE.2005.85.

[60] Gerhard Friedrich, Maria G. Fugini, Enrico Mussi, Barbara Pernici, and Gaston Tagni. Exception handling for repair in service-based processes. *IEEE Transactions on Software Engineering*, 36(2):198–215, 2010. DOI: 10.1109/TSE.2010.8.

[61] Zvi Galil and Giuseppe F. Italiano. Data structures and algorithms for disjoint set union problems. *ACM Computing Surveys*, 23(3):319–344, 1991. DOI: 10.1145/116873.116878.

[62] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN: 0-201-63361-2.

[63] Rob J. van Glabbeek and W. Peter Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, 43(3):555–600, 1996. DOI: 10.1145/233551.233556.

[64] Jim Gray. Why do computers stop and what can be done about it? In *Proceedings of the 5th symposium on Reliability in Distributed Software and Database Systems*. Los Angeles, CA, USA, 1986, pages 3–12.

[65] Lars Grunske, Bernhard Kaiser, and Yiannis Papadopoulos. Model-driven safety evaluation with state-event-based component failure annotations. In *Proceedings of the 8th international conference on Component-Based Software Engineering, CBSE 2005*. Springer-Verlag, 2005, pages 33–48. ISBN: 3-540-25877-9, 978-3-540-25877-3. DOI: 10.1007/11424529_3.

[66] Joaquin Guillén, Javier Miranda, Juan Manuel Murillo, and Carlos Canal. A service-oriented framework for developing cross cloud migratable software. *Journal of Systems and Software*, 86(9):2294 –2308, 2013. ISSN: 0164-1212. DOI: 10.1016/j.jss.2012.12.033.

[67] Reza Hajisheykhi, Ali Ebnenasir, and Sandeep S. Kulkarni. UFIT: a tool for modeling faults in UPPAAL timed automata. In *NASA Formal Methods, 7th international symposium, NFM 2015, Pasadena, CA, USA, april 27-29, 2015, Proceedings*. Klaus Havelund, Gerard Holzmann, and Rajeev Joshi, editors. Springer, 2015, pages 429–435. ISBN: 978-3-319-17524-9. DOI: 10.1007/978-3-319-17524-9_32.

[68] Mohammad Hamdaqa, Tassos Livogiannis, and Ladan Tahvildari. A reference model for developing cloud applications. In *Proceedings of the 1st international conference on Cloud Computing and Services Science, CLOSER 2011*. SciTePress, 2011. ISBN: 978-989-8425-52-2. DOI: 10.5220/0003393800980103.

[69] Pascal Hirmer, Uwe Breitenbücher, Tobias Binz, and Frank Leymann. Automatic topology completion of TOSCA-based cloud applications. In *44. Jahrestagung der Gesellschaft für Informatik, INFORMATIK 2014*. E. Plödereder, L. Grunske, E. Schneider, and D. Ull, editors. Volume 232. In LNI. GI, 2014, pages 247–258.

[70] Paola Inverardi and Massimo Tivoli. Automatic synthesis of modular connectors via composition of protocol mediation patterns. In *Proceedings of the 35th international conference on Software Engineering, ICSE 2013*. IEEE Press, 2013, pages 3–12. DOI: 10.1145/258077.258078.

[71] Bart Jacobs and Jan Rutten. A tutorial on (co)algebras and (co)induction. *EATCS bulletin*, 62:62–222, 1997.

[72] Einar B. Johnsen, Olaf Owe, E. Munthe-Kaas, and Jüri Vain. Incremental fault-tolerant design in an object-oriented setting. In *Proceedings of the 2nd Asia-Pacific conference on Quality Software, APAQS 2001*. IEEE Computer Society, 2001, pages 223–. ISBN: 0-7695-1287-9.

[73] Juju. DevOps distilled. URL: https://juju.ubuntu.com.

[74] Bernhard Kaiser, Peter Liggesmeyer, and Oliver Mäckel. A new component concept for fault trees. In *Proceedings of the 8th australian workshop on Safety Critical Systems and Software, SCS 2003*. Australian Computer Society, Inc., 2003, pages 37–46. ISBN: 1-920-68215-5.

[75] Matthias Klusch, Benedikt Fries, and Katia Sycara. OWLS-MX: a hybrid semantic web service matchmaker for OWL-S services. *Web Semantics*, 7(2):121–133, 2009. ISSN: 1570-8268. DOI: 10.1016/j.websem.2008.10.001.

[76] Woralak Kongdenfha, Hamid Reza Motahari-Nezhad, Boualem Benatallah, Fabio Casati, and Regis Saint-Paul. Mismatch patterns and

adaptation Aspects: a foundation for rapid development of web service adapters. *IEEE Transactions on Services Computing*, 2(2):94–107, 2009. ISSN: 1939-1374. DOI: 10.1109/TSC.2009.12.

[77] Oliver Kopp, Tobias Binz, Uwe Breitenbücher, and Frank Leymann. BPMN4TOSCA: a domain-specific language to model management plans for composite applications. In *Business Process Model and Notation: 4th international workshop, BPMN 2012, Vienna, Austria, September 12-13, 2012. Proceedings*. Jan Mendling and Matthias Weidlich, editors. Springer, 2012, pages 38–52. DOI: 10.1007/978-3-642-33155-8_4.

[78] Oliver Kopp, Tobias Binz, Uwe Breitenbücher, and Frank Leymann. *Winery – A modeling tool for TOSCA-based cloud applications*. In. *Service-Oriented Computing: 11th international conference, ICSOC 2013, Berlin, Germany, December 2-5, 2013, Proceedings*. Samik Basu, Cesare Pautasso, Liang Zhang, and Xiang Fu, editors. Springer, 2013, pages 700–704. DOI: 10.1007/978-3-642-45005-1_64.

[79] Saul Aaron Kripke. Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16(1963):83–94, 1963. DOI: 10.1002/malq.19630090502.

[80] Leslie Lamport. ''sometime'' is sometimes "not never": on the temporal logic of programs. In *Proceedings of the 7th ACM symposium on Principles of Programming Languages*. In PoPL. ACM, 1980, pages 174–185. ISBN: 0-89791-011-7. DOI: 10.1145/567446.567463.

[81] Rogério de Lemos and José Luiz Fiadeiro. An architectural support for self-adaptive software for treating faults. In *Proceedings of the 1st workshop on Self-healing Systems, woss 2002*. ACM, 2002, pages 39–42. ISBN: 1-58113-609-9. DOI: 10.1145/582128.582136.

[82] Frank Leymann. Cloud computing. *it — Information Technology, Methoden und innovative anwendungen der informatik und informationstechnik*, 53(4):163–164, 2011. DOI: 10.1524/itit.2011.9070.

[83] Peter Liggesmeyer and Martin Rothfelder. Improving system reliability with automatic fault tree generation. In *Proceedings of the the 28th annual international symposium on Fault-Tolerant Computing, ftcs 1998*. IEEE Computer Society, 1998, pages 90–99. ISBN: 0-8186-8470-4. DOI: 10.1109/FTCS.1998.689458.

[84] Farzad Mahdikhani, Mahmoud Hashemi, and Marjan Sirjani. QoS aspects in web services compositions. In *Service-Oriented System Engineering, 2008. SOSE'08. IEEE international symposium on*. IEEE, 2008, pages 239–244. DOI: 10.1109/SOSE.2008.39.

[85] Jorge Martinez-Gil, Ismael Navas-Delgado, and Jose Aldana-Montes. MaF: An ontology matching framework. *Journal of Universal Computer Science*, 18(2):194–217, 2012. DOI: 10.3217/jucs-018-02-0194.

[86] Peter M. Mell and Timothy Grance. SP 800-145. The NIST Definition of Cloud Computing. Technical report. Gaithersburg, MD, United States, 2011.

[87] Sonia Ben Mokhtar, Davy Preuveneers, Nikolaos Georgantas, Valérie Issarny, and Yolande Berbers. EASY: Efficient semantic service discovery in pervasive computing environments with QoS and context support. *Journal of Systems and Software*, 81(5):785–808, 2008. ISSN: 0164-1212. DOI: 10.1016/j.jss.2007.07.030.

[88] Hamid Reza Motahari Nezhad, Guang Yuan Xu, and Boualem Benatallah. Protocol-aware matching of web service interfaces for adapter development. In *Proceedings of the 19th international conference on World Wide Web, WWW 2010*. ACM, 2010, pages 731–740. DOI: 10.1145/1772690.1772765.

[89] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989. DOI: 10.1109/5.24143.

[90] Naoyuki Nagatou and Takuo Watanabe. A model-checking based approach to robustness analysis of procedures under human-made faults. In *Asia-Pacific Business Process Management: second Asia-Pacific conference, AP-BPM 2014, Brisbane, QLD, Australia, july 3-4, 2014. Proceedings*. Chun Ouyang and Jae-Yoon Jung, editors. Springer International Publishing, 2014, pages 117–131. DOI: 10.1007/978-3-319-08222-6_9.

[91] Sam Newman. *Building microservices*. O'Reilly Media, Inc., 2015.

[92] OASIS. Cloud Application Management for Platforms (CAMP) Version 1.1. URL: http://docs.oasis-open.org/camp/camp-spec/v1.1/camp-spec-v1.1.pdf. 2014.

[93] OASIS. Open Component Service Architectures (Open-CSA). URL: http://www.oasis-opencsa.org/specifications. 2007.

[94] OASIS. Topology and Orchestration Specification for Cloud Applications. URL: http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.pdf. 2013.

[95] OASIS. Topology and Orchestration Specification for Cloud Applications (TOSCA) Primer. URL: http://docs.oasis-open.org/tosca/tosca-primer/v1.0/tosca-primer-v1.0.pdf. 2013.

[96] OMG. Service Oriented Architecture Modeling Language (SOA-ML). URL: http://www.omg.org/spec/SoaML/1.0.1/. 2012.

[97] Open Grid Forum. Open Cloud Computing Interface (OCCI). URL: http://occi-wg.org/about/specification/. 2013.

[98] Open Mashup Alliance. Enterprise Mashup Markup Language (EMML). URL: https://en.wikipedia.org/wiki/EMML. 2011.

[99] Declan O'Sullivan and David Lewis. Semantically driven service interoperability for pervasive computing. In *Proceedings of the 3rd ACM international workshop on Data Engineering for Wireless and Mobile Access, MobiDe 2003*. ACM, 2003, pages 17–24. ISBN: 1-58113-767-2. DOI: 10.1145/940923.940927.

[100] Claus Pahl. Containerization and the PaaS cloud. *IEEE Cloud Computing*, 2(3):24–31, 2015. DOI: 10.1109/MCC.2015.51.

[101] Claus Pahl and Brian Lee. Containers and clusters for edge cloud architectures – a technology review. In *Proceedings of the 2015 3rd international conference on Future Internet of Things and Cloud, ficloud 2015*. IEEE Computer Society, 2015, pages 379–386. DOI: 10.1109/FiCloud.2015.35.

[102] Claus Pahl, Li Zhang, and Frank Fowley. Interoperability standards for cloud architecture. In *Proceedings of the 3rd international conference on Cloud Computing and Services Science, CLOSER 2013, Aachen, Germany, 8-10 May, 2013*. SciTePress, 2013. ISBN: 978-989-8565-52-5. DOI: 10.5220/0004366901230126.

[103] Dana Petcu, Georgiana Macariu, Silviu Panica, and Ciprian Crăciun. Portable cloud applications - from theory to practice. *Future Generation Computer Systems*, 29(6):1417 –1430, 2013. DOI: 10.1016/j.future.2012.01.009.

[104] Puppet. Puppet labs. URL: https://puppetlabs.com.

[105] Wang Qiang, Lei Yan, Simon Bliudze, and Mao Xiaoguang. Automatic fault localization for BIP. In *Dependable Software Engineering: Theories, Tools, and Applications, SETTA 2015, 1st international symposium, Nanjing, China, November 4-6, 2015, Proceedings*. Xuandong Li, Zhiming Liu, and Wang Yi, editors. Springer, 2015, pages 277–283. ISBN: 978-3-319-25942-0. DOI: 10.1007/978-3-319-25942-0_18.

[106] Ralf H. Reussner, Steffen Becker, and Viktoria Firus. Component Composition with Parametric Contracts. In *Proceedings of Net.ObjectDays*, 2004, pages 155–169.

[107] Davide Sangiorgi. *Introduction to bisimulation and coinduction*. Cambridge University Press, New York, NY, USA, 2011. ISBN: 1107003636, 9781107003637.

[108] Jacopo Soldani. Matching cloud services with TOSCA. Master Thesis, University of Pisa, July 2013.

[109] Jacopo Soldani, Tobias Binz, Uwe Breitenbücher, Frank Leymann, and Antonio Brogi. ToscaMart: a method for adapting and reusing cloud applications. *Journal of Systems and Software*, 113:395–406, 2016. ISSN: 0164-1212. DOI: 10.1016/j.jss.2015.12.025.

[110] Van Tran, Jacky Keung, Anna Liu, and Alan Fekete. Application migration to cloud: A taxonomy of critical factors. In *SECLOUD: proceedings of the 2nd International Workshop on Software Engineering for Cloud Computing*. ACM, 2011, pages 22–28. DOI: 10.1145/1985500.1985505.

[111] W3C. Unified Service Description Language (USDL). URL: http://www.w3.org/2005/Incubator/usdl/XGR-usdl-20111027/. 2011.

[112] Tim Waizenegger, Matthias Wieland, Tobias Binz, Uwe Breitenbücher, Florian Haupt, Oliver Kopp, Frank Leymann, Bernhard Mitschang, Alexander Nowak, and Sebastian Wagner. Policy4TOSCA: A policy-aware cloud service provisioning approach to enable secure cloud computing. In *On the move to meaningful internet systems 2013: OTM confederated international conferences CoopIS, DOA-Trusted Cloud, and ODBASE, Graz, Austria, September 9-13, 2013. Proceedings*. Robert Meersman, Hervé Panetto, Tharam Dillon, Johann Eder, Zohra Bellahsene, Norbert Ritter, Pieter De Leenheer, and Deijing Dou, editors. Springer, 2013, pages 360–376. DOI: 10.1007/978-3-642-41030-7_26.

[113] Johannes Wettinger, Vasilios Andrikopoulos, Steve Strauch, and Frank Leymann. Enabling dynamic deployment of cloud applications using a modular and extensible paas environment. In *Proceedings of the 6th ieee international conference on Cloud Computing, CLOUD 2013*, 2013, pages 478–485. DOI: 10.1109/CLOUD.2013.68.

[114] Johannes Wettinger, Michael Behrendt, Tobias Binz, Uwe Breitenbücher, Gerd Breiter, Frank Leymann, Simon Moser, Isabell Schwertle, and Thomas Spatzier. Integrating configuration management with model-driven cloud management based on TOSCA. In *Proceedings of the 3rd international conference on Cloud Computing and Service Science, CLOSER 2013, 8-10 May 2013, Aachen, Germany*. SciTePress, 2013. DOI: 10.5220/0004376204370446.

[115] Xie Xiong and Zhang Weishi. The current state of software component adaptation. In *Proceedings of the 1st international conference on Semantics, Knowledge and Grid, SKG 2005.* 2005, pages 103–103. DOI: 10.1109/SKG.2005.123.