Alma Mater Studiorum - Università di Bologna

DOTTORATO DI RICERCA IN INFORMATICA
Ciclo: XXVI

Settore Concorsuale di afferenza: 01/B1
Settore Scientifico disciplinare: INF01

Automatic Deployment of Applications in the Cloud

Presentata da: Tudor A. Lascu

Coordinatore Dottorato:	Relatore:
Maurizio Gabbrielli	Gianluigi Zavattaro

Esame finale anno 2014

A Paola e Federico

Abstract

In distributed systems like clouds or service oriented frameworks, applications are typically assembled by deploying and connecting a large number of heterogeneous software components, spanning from fine-grained packages to coarse-grained complex services. The complexity of such systems requires a rich set of techniques and tools to support the automation of their deployment process. By relying on a formal model of components, a technique is devised for computing the sequence of actions allowing the deployment of a desired configuration. An efficient algorithm, working in polynomial time, is described and proven to be sound and complete. Finally, a prototype tool implementing the proposed algorithm has been developed. Experimental results support the adoption of this novel approach in real life scenarios.

Ringraziamenti

A Paola, la mia compagna di vita, va il primo e più grande grazie. L'appoggio che mi ha dato in questi anni è stato essenziale. È stata la prima persona a farmi capire che la cosa più importante è stare bene con se stessi. La vita con te e Federico è piena di gioia.

Per quanto riguarda l'ambiente universitario, sono tante le persone cui devo molto. Prima di tutto, il Professor Cosimo Laneve per aver creduto in me ed avermi convinto che ero in grado di intraprendere il percorso di dottorato. Senza di lui non l'avrei fatto e gliene sono riconoscente.

Ringrazio i Professori Gabbrielli, Martini e Sangiorgi che, abbinando un grande valore umano ad un grande valore professionale, mi hanno aiutato a superare i momenti di difficoltà del mio percorso.

Ringrazio il Professor Gianluigi Zavattaro che ha accettato di "adottarmi scientificamente" e che mi ha sempre accolto con un sorriso.

Ringrazio la Professoressa Vania Sordoni che è sempre stata una figura amica, a partire dalla stesura della tesi di laurea in poi. È bello passare qualche volta a fare quattro chiacchere in simpatia.

Ringrazio i miei compagni di sventura Ornela, Andrea, Giulio, Andrea B., Armir e Nicolò che nel corso delle scuole estive e dei corsi di dottorato sono stati un gruppo simpatico ed affiatato. In particolare, un grosso grazie a Giulio per la compagnia e le innumerevoli chiaccherate alla macchina del caffè.

Ad Elena devo di avermi mostrato che si può fare ricerca con passione ed avere una famiglia, allo stesso tempo.

A Jacopo devo di avermi mostrato cosa vuol dire determinazione e continuità (e puntualità). Grazie anche per avermi s{u,o}pportato come collega di lavoro gomito a gomito.

A Sara devo di tener presente sempre e comunque che siamo prima di tutto esseri umani.

Grazie a tutti i ragazzi dell'underground, vecchie e nuove leve, per la simpatia che mi hanno sempre mostrato: Ivan, Marco, Luca, Saverio, Valeria, Silvio, Roberto, Francesco.

Grazie a Miriam, Daniela, Paolo, Lucia e Vito per aver contribuito a dare una certa atmosfera che mi mancherà.

Grazie ad alcuni ex compagni di studio, Andrea Bagnacani, Matteo Acerbi e Michele Alberti, che hanno ancora voglia di passare a trovarmi.

Ringrazio i membri del progetto Aeolus, in particolare il gruppo di Parigi che mi ha ospitato per il periodo all'estero: Roberto Di Cosmo, Stefano Zacchiroli, Michael Lienhardt, Jakub Zwolakowski, Pietro Abate e Ralf Treinen. Il Professor Roberto Di Cosmo mi ha mostrato che si può essere brillanti con semplicità e umiltà. Grazie a Zack per la disponibilità e per avermi aiutato a cavarmela nella vita di tutti giorni a Parigi. Grazie a Pietro per la pazienza e per avermi svelato l'esistenza di `ocamlbuild`. Grazie a Michael per le chiaccherate durante le pause e per aver sempre cercato di stimolarmi a lavorare, anche quando non ne avevo voglia.

Grazie, infine, ai Professori Jean-Bernard Stefani e Di Cosmo per aver letto con attenzione la tesi ed avermi inviato dei commenti utili.

Per quanto riguarda il mondo "fuori" da quello accademico, ringrazio: Alina, Andrei e Marco; gli amici "da sempre" Paolo, Michele, Giacomo e Stefano; Cristiana e Gabriel; Steve e Beverly; Alessandro; Matteo; Giuliano; Fra e Mari; Camillo; Cecilia. Persone care che ci sono sempre e sempre ci saranno.

Un ultimo caloroso pensiero va ad alcune persone che significano tanto per me e che sono venute a mancare, ma non per questo meno presenti: Mariolina, Pierre e mio papà, colonna portante della mia vita.

Contents

I Preface 1

1 Introduction 3

II Background 6

2 Scenario 7

3 State of the art 9

3.1 Academia 11

3.2 Industry 19

4 Elements of planning theory 27

III Model 33

5 The original Aeolus model 35

5.1 Decidability and complexity 43

5.2 Other component models 45

IV Deployment planning 48

6 Theory 51

6.1 Aeolus$^-$ model & problem statement . . . 51
6.2 Technique . . . 56
6.2.1 Reachability analysis . . . 61
6.2.2 Abstract planning . . . 65
6.2.3 Plan synthesis . . . 72
6.2.4 Heuristics . . . 80
6.3 Formal analysis . . . 82
6.3.1 Soundness & completeness . . . 82
6.3.2 Computational complexity . . . 88

7 Practice 93

7.1 METIS: a deployment planner . . . 93
7.2 Validation . . . 95

V Conclusion 113

8 Future directions 115

8.1 Integration in the Aeolus toolchain . . . 115
8.2 Conflicts . . . 117
8.3 Capacity constraints . . . 120
8.4 Heuristics . . . 121
8.5 Reconfigurations . . . 121
8.6 Restrictions . . . 122

9 Concluding remarks 123

References 131

Part I

Preface

Chapter 1

Introduction

Deploying software component systems is becoming a critical challenge, especially due to the advent of *cloud computing* technologies that make it possible to quickly run complex distributed software systems on-demand on a virtualized infrastructure, at a fraction of the cost which was necessary just a few years ago. When the number of software components, needed to run an application, grows and their interdependencies become too complex to be manually managed, it is necessary for the system administrator to use high-level languages for specifying the system's requirements, and then rely on tools that automatically synthesize the low-level deployment actions necessary to actually realize a correct and complete system configuration that satisfies such requests.

Automation is thus a key ingredient for a wide adoption of cloud facilities. It appears at multiple levels ranging from the installation of packages to scaling computing power (like increasing the number of virtual machines).

In order to deploy an application one needs to specify a sequence of actions like creation/deletion of components, wiring of components (component functionalities), and internal steps to be carried out by each component employed. Moreover, the order in which these actions are to be performed is crucial as it ensures the correctness of all intermediate configurations that the system undergoes. Such a sequence of actions is called a *deployment plan*.

Finding suitable techniques for automatically generating a deployment plan for

complex systems, assembled from a large number of interconnected components, is a serious challenge. This is the goal of the thesis and constitutes the main contribution of this dissertation. The work has been developed as part of the Aeolus research project [1] [1]. The problem is cast in the Aeolus component model [30], specifically tailored to describe uniformly both fine grained software entities, like packages in a Linux distribution, and coarse grained ones, like services, obtained as composition of distributed and properly connected sub-services.

A novel approach for the automatic synthesis of deployment plans has been developed. The technique is shown to be correct, by proving its soundness and completeness, and efficient (of polynomial computational complexity). Moreover, viability in practice of the proposed technique is assessed by means of a proof of concept implementation, validated against standard planning techniques.

Thesis structure. Part II gives an overview of the context for this work. Chapter 2 describes the typical usage scenario, Chapter 3 summarizes the state of the art solutions from both academia and industry, Chapter 4 provides some basic elements of planning theory (that will be of use in the validation part).

Part III describes the original Aeolus component model and reports formal results proving the impossibility to find efficient solutions to the problem of interest, in the general case. In the end, some related component models are recalled.

Part IV is the key part, dedicated to the main contribution of this dissertation, first focusing on the theoretical aspects (Chapter 6) and then on the practical issues (Chapter 7). Chapter 6 starts by presenting Aeolus^{-}, a meaningful restriction of the original model, that allows us to devise an efficient algorithm to deal with the problem of interest. The formal statement of the *deployment problem*, the one we aim to solve, is given. It then continues with the description of the technique developed to tackle the deployment problem together with formal results for its soundness, completeness and efficiency. Chapter 7 deals with the presentation of METIS, a prototype tool, implementing the devised technique, and its validation.

[1] Project ANR-2010-SEGI-013-01.

Part V closes the dissertation by outlining future directions of development for the presented approach (Chapter 8) and by drawing some concluding remarks (Chapter 9).

Part II

Background

Chapter 2

Scenario

In the cloud era, deploying a complex application on commodity (physical or virtual) machines is becoming more and more a common task. This is due to many different reasons such as cost-effectiveness, scalability, etc... The *elastic computing* paradigm enables rapidly adapting an applications' needs to the real usage. It might be cheaper to rely on commodity hardware instead of buying and maintaining it in-house.

In the current setting every organization that needs to perform often this task, typically has a team of experts that establishes how the different components are to be installed and connected together. That is, they find a sequence of actions, a deployment plan, that when performed, permits to achieve the desired system. This part of the work is usually performed by hands with "paper and pencil". The deployment process is then automated by coding it in custom scripts. This approach, however, is effective only if the architecture of the system is decided once and for all. Todays' applications, however, are expected to change at a very high pace as it is common practice to switch to a different service delivering the same required functionality. In fact, for the same functionality different competitors show up on the market everyday and can quickly become appealing. If the system is subject to change the "by-hands approach" does not scale, resulting in a lot of time spent patching the custom scripts to adapt the deployment plan to the new component. This is rather unsatisfactory as a business process and it is natural to ask for a better solution.

The Aeolus project aims to develop techniques and tools, ground on solid scientific bases, to enable simplifying the management of systems/applications to be put in production in the cloud. One of the key ingredients to reach this ambitious goal is a suitable technique to automate the deployment process.

As an example of a possible scenario, one can consider the deployment of WordPress, a popular blog platform. First, an installed Apache web server is needed to be able to install WordPress. In order to activate the latter one must first ensure that all services required are up and running and that they are all properly connected. Bringing it in production requires also to activate the associated service. WordPress, for instance, must connect to an active MySQL node. In fact, WordPress cannot be started before MySQL is running. This is precisely the kind of temporal dependencies taken into account by a deployment plan. Such a plan for this basic example would specify the following steps: first, install and activate an Apache server; install WordPress; install and activate a MySQL instance; finally, connect Wordpress to to the MySQL node and activate WordPress.

Chapter 3

State of the art

Deployment automation is among the key ingredients of the "cloud promise". In fact, the last years have witnessed a constant rise in the interest towards automation of the process for managing a system in the cloud, both in industry and academia. This is testified by many efforts from both worlds to bridge the gap from the traditional/custom way of dealing with the problem to a rich set of techniques and tools enabling a higher level of automation. Managing the installation of an application in the cloud is a process crossing many related areas such as system's deployment, configuration and management.

Currently, developing an application for the cloud is accomplished by relying on either of the following service models: Infrastructure as a Service (IaaS) or Platform as a Service (PaaS). The aim of the former is to provide a set of low-level resources forming a "bare" computing environment like CPUs, memory, network, etc . . . The latter, instead, is meant to provide a full development environment where some middleware services are already accessible (operating system, development kit, runtime libraries, etc . . .).

For IaaS, at the beginning the intended usage scenario was the following: the developer would pack the whole software stack into a virtual machine, containing the application and all its dependencies; the virtual machine would then be hosted on an virtual/physical machine on the provider's cloud. This paradigm however is limited to cases in which the application is not subject to frequent change. In case this does

not apply, the cost of rebuilding from scratch the virtual machine with the whole software stack can become a heavy burden. Another deployment approach, based on IaaS and gaining more and more credit, is the one put forth by the DevOps [3] community. Following this approach an application is developed by assembling available components that serve as the basic building blocks. This emerging approach works thus in a bottom-up direction. From individual component descriptions and recipes for installing them, an application is built as a composition of these recipes. The latter may be seen as deployment plans for individual components and the "global" deployment plan becomes thus the composition of individual ones.

In the PaaS setting, instead, applications are directly written in a programming language supported by the framework offered by the provider, and then "pushed" to the cloud. It is then up to the provider to set up the necessary run-time environment to execute the newly created application. Almost all details of the deployment are handled automatically. The PaaS approach seems promising, as it lifts the level of abstraction, but at the moment the solutions it provides are limited and thus does not represent a valid alternative for the scenario taken into account by our work. In fact, the high-level of automation comes at the (high) price of little flexibility in choosing the components that the developer may use. First of all, the choice of the programming language to employ is restricted to the ones supported by the specific PaaS provider. Moreover, the application code must conform to specific APIs. Google App Engine [4], one of the most successful products in this setting, supports only applications written in Java and Python [1] and in the Java code, threads are not allowed. Another example of the lack of flexibility in the PaaS world is given by Windows Azure [11] that works only with applications built on proprietary technologies. Moreover, the PaaS setting can be seen as a "middleware as a service" solution. Application stacks are thus limited by the middleware services supplied by the PaaS provider. This makes it unfit, at least at present time, for the high degree of customization demanded by ordinary application stacks.

A third way to deal with deploying applications in the cloud is the one employed

[1] Support for Go and PHP language is experimental.

by so called *holistic* frameworks, such as TOSCA [64, 19, 76] and Blueprints [67]. This is a model-driven approach where an application is defined in terms of a high level description. A *projection* process is thus enabled where the deployment plan for the full application is generated in a top-down way. [2] From the private sector, among the others adopting this approach, we can cite IBM SmartCloud Orchestrator [48].

As we will detail in Section 6.2, the approach that we propose shares some commonalities with the above ones and it might actually be conceived as an intermediate way between the full bottom-up and top-down approaches. The description of the application is a high-level one but the deployment plan is inferred from a declarative description of individual components, forming the basic building blocks.

In the following we review state of the art solutions by discussing first works from academia and then tools made available in the industry.

3.1 Academia

Engage

Engage [34] is a deployment management system. Throughout the paper the term *resource* is used as a synonym of *component*. Every resource is represented by two parts: a declarative one, the *type*, and an implementation part, the *driver*. The former is employed to statically verify deployment properties and to generate the deployment plan, while the latter, implemented in a specific language, provides all the required low-level actions to install and manage the resource's life cycle. A notion of hierarchy is introduced by employing three kinds of dependencies: `Inside` models nesting of resources (like a program running into an application server); `Env` models local dependencies, that is resources that the current one requires to find on the same physical or virtual machine (like a program needing a Java execution environment); `Peer` models dependencies to resources possibly deployed anywhere else (one has to look inside and outside the machine of the resource under consideration).

[2] In order to achieve this some form of recipe for the deployment of the bottom level components is obviously necessary.

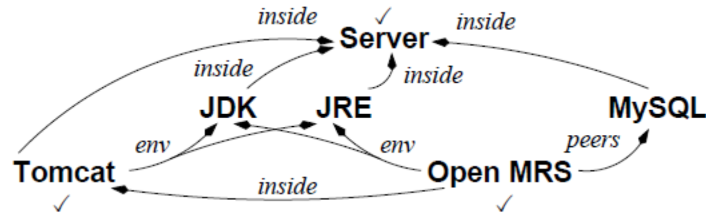


Figure 3.1: Example of hypergraph generated by Engage.

The workflow of the Engage framework is the following. There is a universe of available resources populated by the community (typically the vendor of an application writes down the resource type and the driver for its product). A user writes a (partial) specification of the system he wants to deploy using the resources available in the universe. This (partial) specification is then fed to Engage that first verifies some correctness properties: it mainly amounts to verify that the union of the three dependency relations is acyclic. This is crucial as we will soon explain. The second step is the generation of a hypergraph where nodes and edges represent respectively resource instances and dependencies (each edge is labelled with the kind of dependency). Hyperedges are used to model disjunction of dependencies: a resource requires a functionality provided by two or more resources.

Figure 3.1 depicts an example of the generated hypergraph. In this example the Tomcat resource requires a Java environment to execute, this can be provided by a JRE or by a JDK as shown by the hyperedge, tagged with *env*, to these two resources.

A topological sort of the hypergraph is used to extract an installation order for the deployment plan of the resource instances in the desired system. The acyclicity of the dependency hypergraph ensures that a topological sort exists, thus guaranteeing that a suitable order can always be found. From the hypergraph a set of Boolean constraints is then generated and given as input to a SAT solver. The solution found by the solver corresponds to a $0 - 1$ assignment to every resource instance, telling us if it needs to be installed or not. This information is then put together with the installation order given by the topological sort of the hypergraph to obtain a full

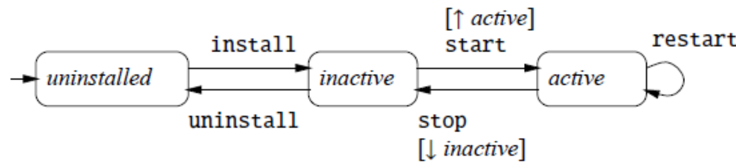


Figure 3.2: Typical state machine associated to a resource driver.

deployment plan.
In the last phase this plan is actually carried out using the driver of each resource. Each resource driver can be seen as a state machine that defines the lifecyle of a resource of that kind. Each driver is thus made of states and transitions. The set of states must contain at least the *uninstalled*, *inactive* and *active* states. Transitions between states take the form of guarded actions $[\downarrow s]\ \alpha$ or $[\uparrow s]\ \alpha$, where s is a boolean condition, the $\downarrow$ and $\uparrow$ arrows define its scope and α is an action that is fired when the transition is taken. The $\uparrow$ arrow means that condition s has to be fulfilled by all the resources upon which the given resource depends on (its upstream dependencies) while the $\downarrow$ arrow means that condition s has to be fulfilled by all the resources that depend on the given one (its downstream dependencies). If and when condition s becomes true then action α is triggered and the transition is fired, otherwise it stays pending.

In Figure 3.2 a typical state machine of a resource r is depicted. Notice that the `start` action and the corresponding transition can be fired only when all the resources that r depends on are already `active`.

The Engage model introduces some important simplifications in order to reach a feasible solution. First of all, conflicts and capacity constraints are not modeled. The acyclicity constraint, crucial to the Engage approach, banishes the possibility of having resources that are mutually dependent, a common case in real-world applications. Moreover, dependencies are between resources, regardless of their current state. i.e. the granularity of dependencies is coarse. The guarded actions are limited in their scope: the only two possibilities are downstream and upstream. Finally, an

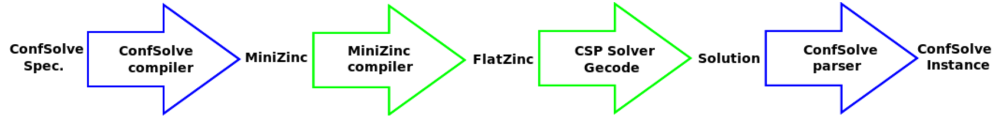


Figure 3.3: ConfSolve workflow.

underlying assumption is that the state machine in a driver forms a strongly connected graph (as each state is required to be reachable from any other in the state machine).

Overall, Engage represents an interesting compromise between applicability and efficiency but for the deployment problem in the cloud.

ConfSolve

The aim of ConfSolve [44, 45] is to define a suitable language for the description of problems related to system's configuration. The sought language should be designed to ease stating such problems, on one side, and enable their translation into *constraint satisfaction problems* (CSPs), on the other. This way one can rely on techniques from the CSP world to tackle the problems of this domain. ConfSolve consists basically into a definition of a *domain specific language* and translation mechanisms to a popular format for the description of CSP problems, namely MiniZinc [62]. The underlying assumption is that specific problems can be naturally modeled (and thus expressed) as constraints over valid configurations.

Figure 3.3 shows the ConfSolve workflow. First a specification is written down in the ConfSolve language. This specification defines both the model and the constraints over it that will specify what is a "valid configuration". The specification is translated by a compiler into the MiniZinc model. This in turn is "flattened"/translated into a FlatZinc model by a third-party compiler. The obtained problem is then fed to a third-party CSP solver, in this case the chosen one is Gecode [36] but this is a completely modular choice (no changes are needed as long as the solver accepts FlatZinc problem definitions). Finally the solution found by the solver is translated back into a ConfSolve instance. This represents a valid con-

figuration that optimizes one or more parameters chosen to define the configuration management problem of interest. As an example one can think of the problem of maximizing the number of (possibly many kinds of) virtual machines per physical one.

The aim of the ConfSolve research was defining a language that would allow to express typical configuration management problems, with direct translation mechanisms to popular formats for stating CSP problems. The ConfSolve language is object oriented and declarative. The ConfSolve specification is a collection of class declarations, enumerations, variables and constraints where the order in which they appear does not matter. Association between objects and names is achieved through the use of reference variables. Declaration `var host as ref Machine;`, for instance, states that `host` is a reference to an object of type `Machine`. Each reference, left unassigned by the user, represents a *decision variable*, that will be instantiated by the solver according to the specified constraints. This constitutes the key idea behind ConfSolve: the system administrator is provided a specific language for defining configuration problems so that solutions can be represented as assignments over some decision variable(s). The language allows using quantification and summation over decision variables in constraints.

```
forall ws in webServers where ws.host != m0 {
  ws.port = 80;
}
```

Above code for example, may be used to require that every element in `webServers`, not running on host `m0`, has port set to `80`. As for summation consider the following code:

```
where foreach (m in machines) {
  sum (r in roles where r.host == m) {
    r.cpu
  } <= m.cpu
}
```

where the above constraint is specifying that for each physical machine if we sum the CPU power of the virtual machines deployed, the total amount does not exceed the capacity bound.
Finally there is also the possibility of writing optimization constraints that instantiate decision variables in way to maximize (or minimize) a given expression.

The major limitation of this approach is that the ConfSolve language models faithfully the problem of optimal provisioning (of virtual machines) rather than focusing on the deployment process. For instance, it does not take into account the wiring aspect, i.e. how to bind the components in use. The steps needed to reach the final (optimal) configuration computed by the solver are also out of scope.

VAMP

VAMP (Virtual Applications Management Platform) [33, 71] is a framework that enhances automatic configuration of a distributed application in the cloud. The framework is made of the following elements: a language to describe the global structure of the application and an environment to manage the runtime deployment of components. The language extends the OVF (Open Virtualization Format) [31] language, that is a proposed standard [3] for a uniform format for applications to be run on virtual machines. The OVF descriptor is an XML file describing the structure of the application. VAMP extends the descriptor with sections that specify the architectural view of the distributed application: interfaces, dependencies and bindings. Listing 3.1 shows a sample *AppArchitectureSection* section contained in the extended OVF descriptor.

The deployment process is then implemented as a decentralized protocol in a self-configuration manner. The approach is interesting but limited for our purposes as it works under the assumption that the dependency graph is acyclic. [4] Another limitation is given by the fact that the developer must specify the virtual machine

[3] Promoted by the Distributed Management Task Force (DMTF).

[4] Dependencies can be optional or mandatory (needed for component activation). It is assumed that there is no cycle among mandatory dependencies.

Listing 3.1: Added section to OVF descriptor

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

in which a given component lives (see line 8 in the above code) : ideally, from our perspective, this is part of low level details that one may not care when defining an application.

A Formal Framework for Component Deployment

In [56] a framework has been developed to formally frame the problem of component deployment. The aim of the work is to model the deployment process of component systems and, based on this, to define a technology-agnostic technique to ensure some correctness properties.

To this purpose a Labeled Transition System (LTS) is defined where states and edges represent, respectively, possible configurations of the system (called Buildboxes) and deployment operations changing the Buildbox.

The properties proved to hold, Well-formedness and Closure, basically amount to the fact that dependency constraints and version compatibility, declared at development time, will be respected during deployment, including possible run-time updates and dynamic component deployment (a.k.a. hot deployment).

There are some key differences in the approach and overall objectives of this work w.r.t. the work presented in this dissertation. First of all, components are seen as monolithic/atomic entities: their internal state representing their behaviour is not part of the model. Each component is considered to be inherently deployable as a singleton independent unit.

Moreover, dependency constraints must specify the name and version of the component that is expected to act as a provider for the required interface. This essential assumption, however, is not reasonable for our purposes as we do not want the developer to constrain to whom a given component is to be bound to access a needed functionality.

Finally, circular dependencies among components are allowed in a weak form as at installation time dependency constraints may be temporarily violated. Correctness is then ensured at run-time. [5] This form of cyclic dependency does not correspond to the one considered in this dissertation as the model adopted in the latter allows to represent circularity of *strong dependencies*, i.e. that must hold at each point in time.

Deployment through planning

Another direction of research is the one leveraging on traditional *planning* techniques and tools coming from the artificial intelligence area. In [16] the problem

[5] If components a and b are mutually dependent ,installing first a and then b is acceptable. This actually corresponds to the notion of *weak requirement* in the Aeolus model, presented in Section 5.

of components' deployment is translated into an instance of planning problems via an encoding into the PDDL language [35] (the *de facto* standard format to define problems in the planning domain). A tool, called Planit, relying on the LPG [39] planner, has been developed.

The model described is based on three kinds of objects: components, machines and connectors. Machines are locations for components and connectors to be deployed. Connectors represent communication channels.

In this work components are seen as atomic entities, their (internal) behaviour not being considered. They are only subject to start, stop and connect to other components (through connectors).

The input to the planner consists in the domain, the current (or initial) state and the goal state. The goal state represents the final desired configuration. In the approach taken in this dissertation, on the other hand, the final configuration is not known in advance but is, rather, computed while trying to achieve the goal state.

The performance evaluation distinguishes between *implicit* and *explicit configuration*. The former requires only that a component be connected, without specifying to whom, while in the latter the information on the identity of the connection must be included. The implicit case is closer to our purposes, as in the cloud world one is typically not interested in which component provides a required functionality as long as there is a component that can provide it. In this case the hardest instance has been one with 40 components, 10 connectors (that may be seen as interfaces) and 10 machines (where components may be located) and it took 412 seconds to compute a plan. Scalability experiments were conducted with up to 120 components.

We will further speculate over the viability of this approach in Section 7.2, dedicated to the validation of the tool developed as part of this thesis.

3.2 Industry

The problem of finding a deployment plan for an application made of many different components shares commonalities with a lot of problems, exhibiting subtle nuances

between them. Each of the proposed tools currently found on the market, addresses problems falling into one (or more) between the following categories:

1. configuration management;
2. service orchestration;
3. interoperability and compatibility;
4. resource provisioning;
5. resource migration.

SmartFrog [41] is Java framework, developed at HP, for managing deployment in a distributed setting. It shares some similarities with the Engage approach as every component has a declarative description and a driver, here called *lifecycle manager*. It lacks, however, a way to use the declarative description to extract some information for the deployment plan or to perform some static checks. DADL (Distributed Application Description Language) [60] is a language extension of SmartFrog that enables to express different kinds of constraints (such as Service Level Agreements SLAs and elasticity). The work, however, focuses on the language aspects. A description of the deployment process is missing and this makes it impossible to relate it to our work.

The Puppet language (and more generally the framework offered by PuppetLabs [50, 69]) and CFEngine [23, 2] are two successful tools aimed at configuration management in a distributed setting. Products that fall in this category are designed to simplify the task to manage the deployment of the same system on large quantities of replicas of the same system. The problem we are taking into account, however, is somehow the opposite one: how to manage the huge amount of possible ways to deploy a given system?

CloudFoundry [75] is a PaaS solution by VMware that allows to select, connect and push to a cloud well defined services (databases, message buses, ...), used as building blocks for writing applications with one of the supported frameworks.

Most of the other efforts fall into the third category as their primary concern is to tackle the problems introduced by *vendor lock-in*. When a company outsources some resources (it may be hardware, platforms or services) to a cloud provider, the access rules to them are specific to the chosen provider. The vendor lock-in problem arises when the company wants to change provider, as there is the need to rewrite the access part on every program using those resources. This is perceived as one of the main difficulties for an ever-growing adoption of the cloud business model. Most of the business offered by cloud brokers is aimed to address this problem. The efforts in this direction are usually made by consortiums, sponsored by private and public funds, and strive to define a standard, upon which to base interoperability and compatibility of resources. Most notable amongst this category are OpenStack [66] and OpenNebula [65].
Another interesting project, (in contact with Aeolus), is CompatibleOne [25], striving to define a universal interface for the description of resource needs.
Finally, there is a homonymous project [15], Aeolus, from RedHat. Its focus is on allowing the definition of a virtual machine (VM) that is exportable to all major cloud providers (Amazon, Rackspace, Heroku, ...). This enables the possibility to migrate a VM to and from cloud providers and also private clouds.

In most of these efforts the user still has to manually put the pieces (components) together in order to obtain the desired system. This is the gap aimed to bridge by the Aeolus project, where the work described in this dissertation constitutes an essential piece. One of the key elements that emerged from the quest for a solution to the problem depicted is the necessity of splitting it in two aspects. The first one is a way to describe declaratively the relationships between the components that form the system, totally ignoring the problem of how to obtain a configuration satisfying the requirements. This declarative part serves two purposes. First, it allows to

statically check some properties of the desired system. Second, it can be used to extract some useful information to find an effective plan for the deployment and reconfiguration of the system. Both of these phases could exploit many different means, ranging from static analysis (e.g. some sort of type inference) to constraint programming techniques to generate a solution.

To the best of our knowledge, conflicts are not taken into account as they introduce a level of difficulty that is hard to cope with. Capacity constraints are also omitted from most of the works listed above.

Table 3.1 contains a summary of the available techniques and tools for managing deployment automation in the cloud. Classification is based on the following categories:

Family whether a framework is based on a top-down (*holistic*) or bottom-up (*DevOps*) technique for generating the deployment plan;

Configuration description the description of an application, written in some given language, may have to be fully specified or not;

Component description the language used to specify individual components;

Projection if the application is entirely described in all the details the framework may support a projection operation that synthesizes a deployment plan;

Platform the platform supported;

Cyclic dependencies indicating whether or not the framework is able to deal with circular dependencies among components.

As it addresses different aspects of the automation challenge, some entries have a field filled by symbol "–" which means that the corresponding classification element may not be applied to that particular tool. Consider, for instance, the second entry, namely ConfSolve. The configuration description is the output returned by the tool and so the entry listing the type and language employed for such a description does

not make sense. The same holds for the *configuration description* entries of Juju, as in this framework there is no way to describe the full configuration of an application.

Other entries have been left with a ? symbol when it was not possible to establish the correctness of the value. This happens for the *cyclic dependencies* entries of HP Cloud Service Automation, as being a proprietary technology we were not able to check if it does or does not support cyclic dependencies among components.

Moreover, ✓ and ✗ symbols are used to denote the fact whether a certain tool, respectively, does or does not deal with/supports the corresponding item. For example, Engage does support *projection* but does not handle *cyclic dependencies*.

There are basically two approaches that stand at opposite sides: the *holistic* and the *DevOps* one, employed to characterize the *Family* entry. In the former, also known as *model-driven* approach, one defines a complete model for the entire application and the deployment plan is then derived in a top-down manner. In the latter approach, instead, to every component is associated some metadata (usually of declarative nature) complemented with some code to drive the component's installation/activation. The matadata part describes essentially the functionalities offered by the component, as well as the functionalities (from other components) required to work properly. Other constraints, like CPU power or amount of RAM, may also be part of the description. The deployment plan is then built in a bottom-up manner by assembling individual components (each component is installed by invoking its specific code).

As of today, most of the industrial products, offered by big companies, such as Amazon, HP and IBM, fall in the holistic approach category. In this context, one prominent work is represented by the TOSCA (Topology and Orchestration Specification for Cloud Applications) standard [64], promoted by the OASIS consortium [63] for open standards. TOSCA proposes an XML-like rich language to describe an application. Most of the above vendors now supports TOSCA specifications.

The most important representative for the DevOps approach is Juju [49], by Canonical (the company developing the Ubuntu Linux distribution). It is based on

the concept of *charm*: the atomic unit containing a description of the required and provided functionalities of a service. This description in form of metadata is coupled with configuration data and *hooks* (basically a collection of binary files necessary for the deployment of the given component). Juju is one of the few projects trying to add an orchestration layer between services. Lately, the Juju team has overcome one of the main limitations of the tool, namely the (heavy) assumption that each service unit must be deployed to a separate machine. This effort, although notable, does not seem to have solved the problem that concerns us because some unnecessary manual intervention is still needed. Consider, for instance, the deployment of WordPress in a basic scenario where its only requirement is to be connected to a MySQL database. One would first deploy WordPress by simply typing *#juju deploy wordpress* and then deploy MySQL by *#juju deploy mysql*. Finally one would have to establish the binding between the two components by entering the following command *#juju add−relation wordpress mysql*. Now, as the metadata (`metadata.yaml` file), part the WordPress charm, contains a require entry on interface *mysql*, provided by MySQL, it is not clear why should we manually create the actual connection among the two components. Moreover, as of today, there is no way to statically detect anomalies such as bringing up a WordPress instance without any prior deployment of a (MySQL) database. This would actually result in a run-time error, to be discovered only after having "successfully" deployed WordPress.
As a final remark notice that usage of Juju is limited to Ubuntu distributions.

The strategy adopted by METIS represents somehow a breed between the DevOps and the holistic approaches. It starts with individual description for each component, as in the DevOps methodology, but the final deployment plan is not the result of assembling different local plans (for each component), but is rather obtained by means of a unitary projection process, typical of the holistic world.

Available Frameworks							
Name	**Family**	**Configuration description**		**Component description**	**Projection**	**Platform**	**Cyclic dependencies**
		type	**language**	**language**			
Engage	DevOps	partial	JSON	JSON	✓	independent	✗
ConfSolve	–	–	ConfSolve	–	–	independent	–
SmartFrog	–	full	SmartFrog DSL	recipe	✗	independent	✗
VAMP	holistic	full	OVF	OVF	✓	independent	✗
Juju	DevOps	–	–	recipe (charm)	✗	Ubuntu	✗
TOSCA	holistic	full	TOSCA	TOSCA	✗	independent	✗
Amazon AWS CloudFormation	holistic	full	JSON	JSON	✗	Amazon (AMI, EC2, S3)	✗
HP Cloud Service Automation	holistic	full	graphical & TOSCA	graphical & TOSCA	✗	private & hybrid cloud	?
IBM SmartCloud Orchestrator	holistic	full	patterns & TOSCA	proprietary & TOSCA	✗	OpenStack	✗
METIS	holistic/DevOps	partial	JSON	JSON	✓	independent	✓

Table 3.1: Available techniques & tools for deployment automation.

Chapter 4

Elements of planning theory

The first approach that comes to mind when facing the problem of dealing with the automatic synthesis of deployment plans is, naturally, *planning*. Planning is a well-established area of the *artificial intelligence* field, devoted to the computation of the actions to be performed in order to reach some final goal state of a dynamic system. In the following we will explain why this is not a suitable approach for our purposes. In order to argue for the need of a specialized approach, presented in this dissertation, some basic elements of planning theory are here recalled. The concepts introduced will also ease the understanding of the validation part (Section 7.2), which is based on encoding the problem addressed herein into a classical planning one.

As a "slogan definition" *planning is the reasoning side of acting* [40]. Starting from a description of the world considered and the possible ways to move from a specific situation to the subsequent one, the aim is to find a way to reach a *goal* situation. A planning problem is specified by a description of the world, modeling a domain of interest, an initial state and a goal state (or more generally a set of goal states).

A dynamic system is defined by means of a state transition system. The problem we are interested in lies in the *classical planning* area. Classical planning refers to planning where the transition system considered meets some restricting conditions such as being deterministic, having implicit notion of time, having no (relevant)

internal dynamics, etc...

In order to provide the formal statement of the planning problem we need first to define the concepts of state, action, plan and domain.

Planning problem

We will start by giving a first general/generic definition of the planning problem. We will see that we need to specify other details in order to fully define this class of problems.

Definition 4.1 (State transition system)**.** *A* state transition system *is a triple* $\Sigma = (\mathcal{S}, \mathcal{A}, \gamma)$*, where:*

- $\mathcal{S}$ *is a finite set of states;*
- $\mathcal{A}$ *is a finite set of actions;*
- $\gamma : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$

Notation. In the following, for clarity, we will sometimes use $s_i \xrightarrow{a_j} s_{i+1}$ in place of $\gamma(s_i, a_j) = s_{i+1}$.

Based on previous definition we can already define the general form of planning problem.

Definition 4.2 (Generic planning problem)**.** *Consider a triple* (Σ, s_0, S_g)*, where:* $\Sigma = (\mathcal{S}, \mathcal{A}, \gamma)$ *is a state transition system,* s_0 *is the initial state and* $S_g \subseteq \mathcal{S}$ *is a set of goal states. The* planning problem *is finding a sequence of actions* $\langle a_1, a_2, \ldots, a_k \rangle$ *in* $\mathcal{A}$ *s.t.* $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \cdots s_{k-1} \xrightarrow{a_k} s_k$ *with* $s_k \in S_g$*.*

In this formulation the planning problem is equivalent to the *graph reachability problem*, where nodes are states and arcs are defined by the state transition function. One should, however, consider that the above one is a conceptual model. A characteristic ingredient of the class of planning problems is the representation of

the input graph/set of states $\mathcal{S}$. An explicit representation is not viable as the number of states is unmanageable even for simple problems. Part of the challenge in planning is, in fact, given by finding a compact representation for the set of states $\mathcal{S}$. In planning problems the set of states is thus provided in implicit form. Just to give the idea, this is achieved by listing the properties that hold in some state and how they are transformed via actions.

The above definition is parametric w.r.t. the way the transition system Σ is specified and this in turn depends on the chosen representation. There are many possible representations available: the *set-theoretic* one where properties are stated with propositional logic, the *classical* one which, instead, relies on first-order logic and finally the *state-variables* one where property modifications by actions are defined by functions mapping variables associated to states into the result value. These representations are equivalent w.r.t. the planning problems that can be modeled. In the following we will detail the *classical representation*, chosen for two reasons: first, it is more compact than the first one; second, its language is closer to PDDL (Planning Domain Definition Language), the language in use by the vast majority of tools from the planning community. As a consequence we obtain an instantiation of the generic planning problem defined in Definition 4.2.

States & actions

We begin by defining the language used by classical planning.

Definition 4.3 (Classical planning language)**.** *The language $\mathcal{L}$ is a* classical planning language *if it is a first-order language s.t. :*

- *the set of predicates and the set of constant symbols are finite;*
- *there are no function symbols.*

Each state is represented by a set of ground atoms of $\mathcal{L}$: an atom p holds in a

state s if and only if $p \in s$. [1] The set of states $\mathcal{S}$ must be finite due to the above restrictions on $\mathcal{L}$.

In the following, sans-serif fonts are used for predicate and constant symbols.

An action a is specified by means of *preconditions* and *effects*. The former define when an action may be applied, while the latter define how a's application affects the current state. Actions are defined as instantiations of operators. Operators can be seen as rules that apply for generic objects of the world considered and each action is the concretization of an operator. For instance, one may have an operator $\mathsf{move}(r, l, m)$, where move is a predicate symbol in $\mathcal{L}$, whose intended meaning is "robot r moves from location l to an adjacent location m". Then, a possible corresponding action would be something like $\mathsf{move}(\mathsf{robot1}, \mathsf{loc2}, \mathsf{loc3})$, where $\mathsf{robot1}$, $\mathsf{loc2}$ and $\mathsf{loc3}$ are constant symbols in $\mathcal{L}$.

We have to specify when an action is enabled, i.e. can be applied in current state. Given a set of literals L, let us denote with L^+ the set of atoms that appear in L and with L^- the set of atoms whose negation is in L. Then for a given action a we can divide preconditions and effects into their positive and negative part, denoted respectively $\mathit{preconditions}^+(a)$, $\mathit{preconditions}^-(a)$ and $\mathit{effects}^+(a)$, $\mathit{effects}^-(a)$.

Definition 4.4 (Applicable action)**.** *An action a is* applicable *in a state s if the following conditions holds:*

- $\mathit{preconditions}^+(a) \subseteq s$, *and*
- $\mathit{preconditions}^-(a) \cap s = \emptyset$.

Applying a to state s is defined by: $\gamma(s, a) \stackrel{def}{=} (s \setminus \mathit{effects}^-(a)) \cup \mathit{effects}^+(a)$.

Domain & planning problem

Based on previous section we can define the domain of a planning problem and provide the actual definition of planning problem, as well as its statement.

[1] Notice that the closed-world assumption is in use: if an atom q does not belong to s then it does not hold in s.

Definition 4.5 (Planning domain). *Let $\mathcal{L}$ be a classical planning language. A* classical planning domain *in $\mathcal{L}$ is a state-transition system $\Sigma = (\mathcal{S}, \mathcal{A}, \gamma)$, where:*

- $\mathcal{S} \subseteq 2^{\{\textit{all ground atoms in } \mathcal{L}\}}$*;*
- *$\mathcal{A}$ is the set of all ground instances of a set $\mathcal{O}$ of operators;*
- $\gamma(s, a) \stackrel{def}{=} \begin{cases} (s \setminus \textit{effects}^-(a)) \cup \textit{effects}^+(a) & \textit{if } a \textit{ is applicable} \\ \perp & \textit{otherwise} \end{cases}$
- *$\mathcal{S}$ is closed under γ, i.e. if γ is defined at (s, a) and $\gamma(s, a) = s'$, then $s' \in \mathcal{S}$.*

We are now ready to define the planning problem and its statement. The statement of a problem may be seen as the way a planning problem is specified in practice. The set $\mathcal{S}$ of states, for example, is not given as is but is the one that can be inferred from a list of operators $\mathcal{O}$.

Definition 4.6 (Classical planning problem). *A* classical planning problem *is a triple $\mathcal{P} = (\Sigma, s_0, g)$, where:*

- *Σ is a classical planning domain;*
- *s_0 is the initial state, in $\mathcal{S}$;*
- *g represents the goal, a set of ground literals;*

The statement *of a planning problem $\mathcal{P} = (\Sigma, s_0, g)$ is $P = (\mathcal{O}, s_0, g)$ where $\mathcal{O}$ is a set of operators.*

Computational complexity

The computational complexity class of planning problems ranges from constant to NEXPTIME-complete according to the representation adopted and the restrictions that may apply in particular cases. Examples of such restrictions are whether or not negative preconditions and/or negative effects are allowed. Negative preconditions

and negative effects simply amount to allow negative atoms to appear in operators' preconditions and effects. Another typical restriction is whether the set of operators $\mathcal{O}$ is fixed in advance or is part of the input. A complete classification of the computational complexity of the planning problem w.r.t. to the restrictions adopted is summarized in [40]. This classification is based on results that appear in [24, 32, 17]. As we will later explain, the encoding of the deployment problem demands for both negative preconditions and effects. As a result, the complexity class of the problem considered in this dissertation is PSPACE. These computational complexity considerations already hint at the fact that a direct encoding of the deployment problem into a generic planning problem might not lead to a viable solution. We will further discuss this issue in Section 7.2, dedicated to the validation of a prototype tool, implementing an ad-hoc planning technique.

Part III

Model

Chapter 5

The original Aeolus model

The component model adopted to frame the deployment problem, called *Aeolus*$^-$, is a restriction of the more complete and complex *Aeolus* model. Current chapter introduces the latter, while the former is presented in Section 6.1.
The Aeolus model has been developed to allow formal reasoning upon typical issues that arise in the process of deploying and reconfiguring a system in the cloud.

In Aeolus a component is described by a declarative specification of its behaviour by means of *states* and *ports*. This information is captured by so-called *component types*: every component belongs to a certain component type. The relevant internal states of components are represented by means of a *finite state automaton* [1] (see Figure 5.1): depending on the current state, components activate *provided* and *required ports*, and get in *conflict* with ports provided by others (in Figure 5.1 active ports are black, while inactive ones are grey). Each port is identified by an interface name. Bindings can be established between provided and required ports with the same interface. Figure 5.1 shows the graphical representation of a typical deployment of the WordPress platform. WordPress requires a Web server providing httpd in order to be installed, and an active MySQL database server in order to be in production. In the example the chosen Web server is Apache2. Notice that Apache2

[1] It is important to notice that automata employed in Aeolus do not represent the internal behavior of components, but rather the effect on the component of an external deployment or reconfiguration actions.

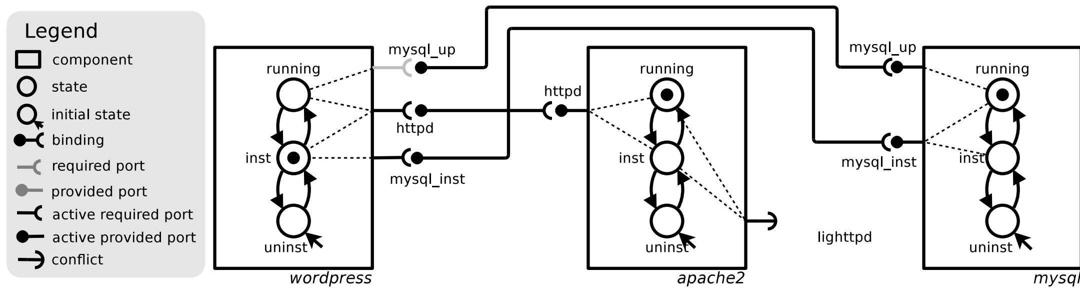


Figure 5.1: Typical Wordpress/Apache/MySQL deployment, modeled in Aeolus.

is not co-installable with other Web servers, such as lighttpd. [2] This constraint is depicted by means of a conflict arrow that is active in states inst and running of the Apache2 component.

At present time the *Aeolus* model is "flat" in the sense that all components live in a single "global" context, are mutually visible, and can connect to each other as long as their ports are compatible. To introduce a notion of hierarchy, different extensions to the original model, enriching it with membranes or boxes, have been envisaged but this part is still ongoing work.

Installing software on a single machine is a process that can already be automated using *package managers*: on Debian for instance, you only need to have an installed Apache server to be able to install WordPress. But bringing it *in production* requires to activate the associated service, which is more tricky and less automated: the system administrator will need to edit configuration files so that WordPress knows the network addresses of an accessible MySQL instance.

Services often need to be deployed on different machines to reduce the risk of failure or due to the limitations on the load they can bear. For example, system administrators might want to indicate that a MySQL instance can only support a certain number of WordPress instances. Symmetrically, a WordPress hosting service may want to expose a reverse web proxy / load balancer to the public and require

[2] Roughly specking, *co-installable* packages are packages that do not conflict. Refer to [57] for the precise definition.

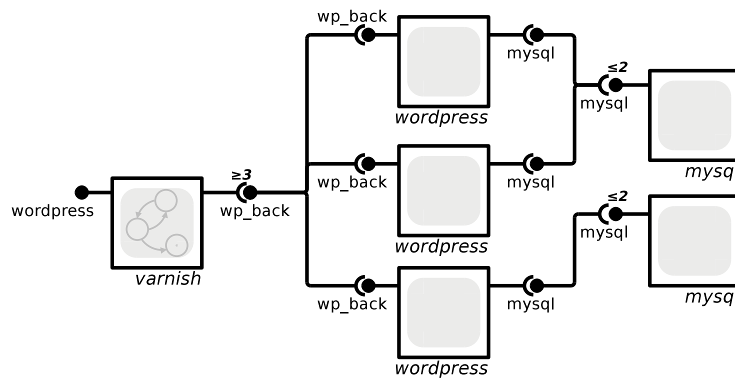


Figure 5.2: A graphical description of the model with redundancy and capacity constraints (internal sate machines and activation arcs omitted for simplicity).

to have a minimum number of *distinct* instances of WordPress available as its back-ends.

To model this kind of situations, Aeolus allows capacity information to be added on provided and required ports of each component: a number n on a provided port indicates that it can fulfill no more than n requirements, while a number n on a required port means that it needs to be connected to at least n provided ports from n *different* components. This information may then be employed by a planner to find an optimal replication of the components to satisfy a user requirement.

As an example, Figure 5.2 shows the modeling of a WordPress hosting scenario where one wants to offer high availability hosting by putting the Varnish reverse proxy / load balancer in front of several WordPress instances, all connected to a shared replicated MySQL database [3]. For a configuration to be correct, the model requires that Varnish is connected to at least 3 (active and distinct) WordPress back-ends, and that each MySQL instance does not serve more than 2 clients.

As a particular case, a 0 constraint on a required port means that no provided port with the same name can be active at the same time; this can be effectively used to model conflicts between components.

[3] All WordPress instances run within separate Apache-s, which have been omitted for simplicity.

Notation. We consider the following disjoint sets: $\mathcal{I}$ for interfaces and $\mathcal{Z}$ for components. We use $\mathbb{N}$ to denote strictly positive natural numbers, $\mathbb{N}_\infty$ for $\mathbb{N}$ plus infinity, and $\mathbb{N}_0$ for $\mathbb{N}$ plus 0.

Terminology. There is a distinction between the concept of *interface* and that of *port*: the latter being an "implementation" of the former. Throughout this work the two terms are sometimes used as synonyms, whenever there is no ambiguity in the given context.

In Aeolus components are modeled as finite state automata indicating the current state and the possible transitions. When a component changes its state, it can also change the ports that it requires from and provides to other components, thus adjusting its behaviour.

Definition 5.1 (Component type)**.** *The set $\mathcal{T}_{flat}$ of* component types *of the Aeolus model, ranged over by $\mathcal{T}_1, \mathcal{T}_2, \ldots$ contains 5-ple $\langle Q, q_0, T, P, D\rangle$ where:*

- *Q is a finite set of states;*
- *$q_0 \in Q$ is the initial state and $T \subseteq Q \times Q$ is the set of* transitions*;*
- *$P = \langle \mathbf{P}, \mathbf{R}\rangle$, with $\mathbf{P}, \mathbf{R} \subseteq \mathcal{I}$, is a pair composed of the set of* provided *and the set of* required *interfaces, respectively;*
- *D is a function from Q to 3-ple in $(\mathbf{P} \mapsto \mathbb{N}_\infty) \times (\mathbf{R} \mapsto \mathbb{N}_0) \times (\mathbf{R} \mapsto \mathbb{N}_0)$.*

Given a state $q \in Q$, the three partial *functions in $D(q)$ indicate respectively the provided, weakly required, and strongly required ports that q activates. The functions associate to the active ports a numerical constraint indicating:*

- *for provided ports, the maximum number of bindings the port can satisfy,*
- *for required ports, the minimum number of required bindings to distinct components,*
  - *if the number is 0, that indicates a conflict, meaning that there should be no other active port with the same name.*

We assume as default constraints ∞ for provided ports (i.e. they can satisfy an unlimited amount of requires) and 1 for required (i.e. one provide is enough to satisfy the requirement). We also assume that the initial state q_0 has no strong demands (i.e. the third function of $D(q_0)$ is empty).

We now define configurations that describe systems composed by components and their bindings. A configuration, ranged over by $\mathcal{C}_1, \mathcal{C}_2, \ldots$, is given by a set of component types, a set of deployed components in some state, and a set of bindings. Formally:

Definition 5.2 (Configuration)**.** *A* configuration $\mathcal{C}$ *is a 4-ple* $\langle U, Z, S, B\rangle$ *where:*

- $U \subseteq \mathcal{T}_{flat}$ *is the* universe *of the available component types;*
- $Z \subseteq \mathcal{Z}$ *is the set of the currently deployed* components*;*
- S *is the component* state description*, i.e. a function that associates to components in* Z *a pair* $\langle \mathcal{T}, q\rangle$ *where* $\mathcal{T} \in U$ *is a component type* $\langle Q, q_0, T, P, D\rangle$*, and* $q \in Q$ *is the current component state;*
- $B \subseteq \mathcal{I} \times Z \times Z$ *is the set of* bindings*, namely 3-ple composed by an interface, the resource that requires that interface, and the resource that provides it; we assume that the two components are distinct.*

Notation. We write $\mathcal{C}[z]$ as a lookup operation that retrieves the pair $\langle \mathcal{T}, q\rangle = S(z)$, where $\mathcal{C} = \langle U, Z, S, B\rangle$. On such a pair we then use the postfix projection operators `.type` and `.state` to retrieve $\mathcal{T}$ and q, respectively. Similarly, given a component type $\langle Q, q_0, T, \langle \mathbf{P}, \mathbf{R}\rangle, D\rangle$, we use projections to (recursively) decompose it: `.states`, `.init`, and `.trans` return the first three elements; `.prov`, `.req` return $\mathbf{P}$ and $\mathbf{R}$; $.\mathbf{P}\texttt{map}(q)$, $.\mathbf{R}_w\texttt{map}(q)$, and $.\mathbf{R}_s\texttt{map}(q)$ return the three elements of the $D(q)$ tuple. When there is no ambiguity we take the liberty to apply the component type projections to $\langle \mathcal{T}, q\rangle$ pairs. *Example*: $\mathcal{C}[z].\mathbf{R}_s\texttt{map}(q)$ stands for the strongly required ports (and their arities) of component z in configuration $\mathcal{C}$ when it is in state q.

We are now ready to formalize the notion of configuration correctness. We consider two distinct notions of correctness: *weak* and *strong*. According to the former,

only weak requirements are considered, while the latter also considers strong ones. Intuitively, weak correctness can be temporarily violated during the deployment of a new component configuration, but needs to be fulfilled at the end; strong correctness, on the other hand, shall never be violated.

Definition 5.3 (Correctness). *Let us consider the configuration $\mathcal{C} = \langle U, Z, S, B \rangle$. We write $\mathcal{C} \models_{req} (z, r, n)$ to indicate that the required port of component z, with interface r, and associated number n is satisfied. Formally, if $n = 0$ all components other than z cannot have an active provided port with interface r, namely for each $z' \in Z \setminus \{z\}$ such that $\mathcal{C}[z'] = \langle \mathcal{T}', q' \rangle$ we have that r is not in the domain of $\mathcal{T}'.\mathbf{P}\mathtt{map}(q')$. If $n > 0$ then the port is bound to at least n active ports, i.e. there exist n distinct components $z_1, \ldots, z_n \in Z \setminus \{z\}$ such that for every $1 \leq i \leq n$ we have that $\langle r, z, z_i \rangle \in B$, $\mathcal{C}[z_i] = \langle \mathcal{T}^i, q^i \rangle$ and r is in the domain of $\mathcal{T}^i.\mathbf{P}\mathtt{map}(q^i)$.*

Similarly for provides, we write $\mathcal{C} \models_{prov} (z, p, n)$ to indicate that the provided port of resource z, with interface p, and associated number n is not bound to more than n active ports. Formally, there exist no m distinct components $z_1, \ldots, z_m \in Z \setminus \{z\}$, with $m > n$, such that for every $1 \leq i \leq m$ we have that $\langle p, z_i, z \rangle \in B$, $S(z_i) = \langle \mathcal{T}^i, q^i \rangle$ and p is in the domain of $\mathcal{T}^i.\mathbf{R}_w\mathtt{map}(q^i)$ or $\mathcal{T}^i.\mathbf{R}_s\mathtt{map}(q^i)$.

The configuration $\mathcal{C}$ is correct *if for each component z in Z, given $S(z) = \langle \mathcal{T}, q \rangle$ with $\mathcal{T} = \langle Q, q_0, T, P, D \rangle$ and $D(q) = \langle \mathcal{P}, \mathcal{R}_w, \mathcal{R}_s \rangle$, we have that $(p \mapsto n_p) \in \mathcal{P}$ implies $\mathcal{C} \models_{prov} (z, p, n_p)$, and $(r \mapsto n_r) \in \mathcal{R}_w$ implies $\mathcal{C} \models_{req} (z, r, n_r)$, and $(r \mapsto n'_r) \in \mathcal{R}_s$ implies $\mathcal{C} \models_{req} (z, r, n'_r)$.*

Analogously we say that it is strong correct *if only the strong requirements are considered: namely, we require $(p \mapsto n_p) \in \mathcal{P}$ implies $\mathcal{C} \models_{prov} (z, p, n_p)$ and $(r \mapsto n_r) \in \mathcal{R}_s$ implies $\mathcal{C} \models_{req} (z, r, n_r)$.*

As our main interest is planning, we now formalize how configurations evolve from one state to another, by means of atomic actions.

Definition 5.4 (Actions). *The set $\mathcal{A}$ contains the following actions:*

- *stateChange(z, q_1, q_2) where $z \in \mathcal{Z}$;*

- $bind(r, z_1, z_2)$ *where* $z_1, z_2 \in \mathcal{Z}$ *and* $r \in \mathcal{I}$;
- $unbind(r, z_1, z_2)$ *where* $z_1, z_2 \in \mathcal{Z}$ *and* $r \in \mathcal{I}$;
- $new(z : \mathcal{T})$ *where* $z \in \mathcal{Z}$ *and* $\mathcal{T} \in \mathcal{T}_{flat}$;
- $del(z)$ *where* $z \in \mathcal{Z}$.

The execution of actions can now be formalized using a labeled transition systems on configurations, which uses actions as labels.

Definition 5.5 (Reconfigurations)**.** *Reconfigurations are denoted by transitions* $\mathcal{C} \xrightarrow{\alpha} \mathcal{C}'$ *meaning that the execution of* $\alpha \in \mathcal{A}$ *on the configuration* $\mathcal{C}$ *produces a new configuration* $\mathcal{C}'$*. The transitions from a configuration* $\mathcal{C} = \langle U, Z, S, B\rangle$ *are defined as follows:*

$$\mathcal{C} \xrightarrow{stateChange(z,q_1,q_2)} \langle U, Z, S', B\rangle \quad \textit{if } \mathcal{C}[z].\texttt{state} = q_1 \textit{ and } (q_1, q_2) \in \mathcal{C}[z].\texttt{trans} \textit{ and } S'(z') = \begin{cases} \langle \mathcal{C}[z].\texttt{type}, q_2\rangle & \textit{if } z' = z \\ \mathcal{C}[z'] & \textit{otherwise} \end{cases}$$

$$\mathcal{C} \xrightarrow{bind(r,z_1,z_2)} \langle U, Z, S, B \cup \langle r, z_1, z_2\rangle\rangle \quad \textit{if } \langle r, z_1, z_2\rangle \notin B \textit{ and } r \in \mathcal{C}[z_1].\texttt{req} \cap \mathcal{C}[z_2].\texttt{prov}$$

$$\mathcal{C} \xrightarrow{unbind(r,z_1,z_2)} \langle U, Z, S, B \setminus \langle r, z_1, z_2\rangle\rangle \quad \textit{if } \langle r, z_1, z_2\rangle \in B$$

$$\mathcal{C} \xrightarrow{new(z:\mathcal{T})} \langle U, Z \cup \{z\}, S', B\rangle \quad \textit{if } z \notin Z,\ \mathcal{T} \in U \textit{ and } S'(z') = \begin{cases} \langle \mathcal{T}, \mathcal{T}.\texttt{init}\rangle & \textit{if } z' = z \\ \mathcal{C}[z'] & \textit{otherwise} \end{cases}$$

$$\mathcal{C} \xrightarrow{del(z)} \langle U, Z \setminus \{z\}, S', B'\rangle \quad \textit{if } S'(z') = \begin{cases} \bot & \textit{if } z' = z \\ \mathcal{C}[z'] & \textit{otherwise} \end{cases} \textit{ and } B' = \{\langle r, z_1, z_2\rangle \in B \mid z \notin \{z_1, z_2\}\}$$

Notice that in the definition of the transitions there is no requirement on the reached configuration: the correctness of these configurations will be considered at the level of deployment run (Definition 5.7).

Also, we observe that there are configurations that cannot be reached through sequences of the actions we have introduced so far. In Figure 5.3, for instance, there is no way for package `a` and `b` to reach the installed state, as each package require the other to be installed *first*. In practice, when confronted with such situations—that can be found for example in FOSS distributions in the presence of

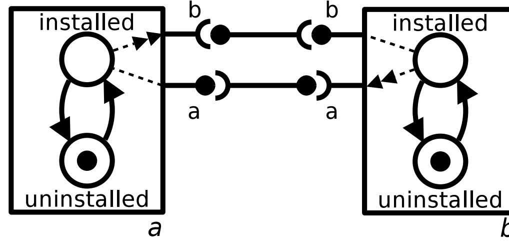


Figure 5.3: On the need of a *multiple state change* action: how to install a and b?

Pre-Depend loops—current tools either perform all the state changes atomically, or abort deployment.

If one wants a planner to be able to propose reconfigurations containing such atomic transitions, one has to introduce the notion of *multiple state change*. [4]

Definition 5.6 (Multiple state change)**.** *A* multiple state change $\mathcal{M} = \{stateChange(z^1, q_1^1, q_2^1), \cdots, stateChange(z^l, q_1^l, q_2^l)\}$ *is a set of state change actions on different component (i.e.* $z^i \neq z^j$ *for every* $1 \leq i < j \leq l$*). We use* $\langle U, Z, S, B\rangle \xrightarrow{\mathcal{M}} \langle U, Z, S', B\rangle$ *to denote the effect of the simultaneous execution of the state changes in* $\mathcal{M}$*: formally,* $\langle U, Z, S, B\rangle \xrightarrow{stateChange(z^1,q_1^1,q_2^1)} \ldots \xrightarrow{stateChange(z^l,q_1^l,q_2^l)} \langle U, Z, S', B\rangle$*.*

Notice that the order of execution of the state change actions does not matter as all the actions are executed on different components.

We can now define a *deployment run*, which is a sequence of actions that transform an initial configuration into a final correct one without violating strong correctness along the way. A deployment run is the output we expect from a planner, when it is asked how to reach a desired target configuration.

[4]This kind of actions are part of the original model because one of its objectives was modeling uniformly both fine-grained components (such as packages) and coarse-grained ones (as services). If one focuses on modeling the latter, however, this kind of action can be ignored in favour of simplicity. This is the approach followed in this thesis, as we will explain in next chapter.

Definition 5.7 (Deployment run)**.** *A* deployment run *is a sequence $\alpha_1 \ldots \alpha_m$ of actions and multiple state changes such that there exist $\mathcal{C}_i$ such that $\mathcal{C} = \mathcal{C}_0$, $\mathcal{C}_{j-1} \xrightarrow{\alpha_j} \mathcal{C}_j$ for every $j \in \{1, \ldots, m\}$, and the following conditions hold:*

configuration correctness *$\mathcal{C}_0$ and $\mathcal{C}_m$ are correct while, for every $i \in \{1, \ldots, m-1\}$, $\mathcal{C}_i$ is strong correct;*

multi state change minimality *if α_j is a multiple state change then there exists no proper subset $\mathcal{M} \subset \alpha_j$, or state change action $\alpha \in \alpha_j$, and correct configuration $\mathcal{C}'$ such that $\mathcal{C}_{j-1} \xrightarrow{\mathcal{M}} \mathcal{C}'$, or $\mathcal{C}_{j-1} \xrightarrow{\alpha} \mathcal{C}'$.*

We now have all the ingredients to define the notion of *achievability*: given an universe of component types, we want to know whether it is possible to deploy at least one component of a given component type $\mathcal{T}$ in a given state q.

Definition 5.8 (Achievability problem)**.** *The* achievability problem *has as input an universe U of component types, a component type $\mathcal{T}$, and a target state q. It returns as output **true** if there exists a deployment run $\alpha_1 \ldots \alpha_m$ such that $\langle U, \emptyset, \emptyset, \emptyset \rangle \xrightarrow{\alpha_1} \mathcal{C}_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_m} \mathcal{C}_m$ and $\mathcal{C}_m[z] = \langle \mathcal{T}, q \rangle$, for some component z in $\mathcal{C}_m$. Otherwise, it returns **false**.*

Remark 5.1. *Notice that the restriction in this decision problem to one component in a given state is not limiting: one can easily encode any given final configuration by adding a dummy provided port enabled only by the desired final states and a dummy component with weak requirements on all such provided ports.*

5.1 Decidability and complexity

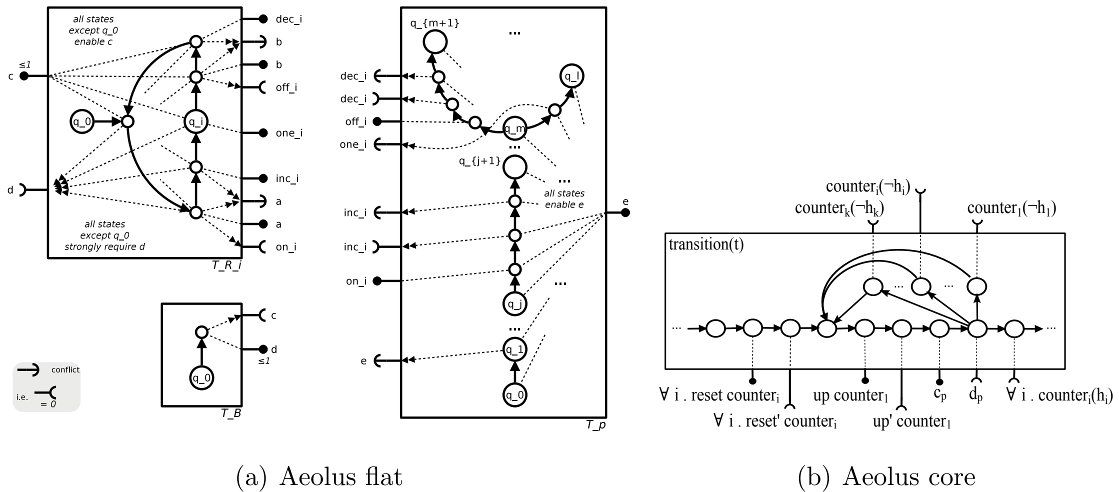
In this section we briefly summarize the formal results established for the achievability problem (Definition 5.8) cast in different variants of the original Aeolus model.

Table 5.1 gives an overview of the results proven in [30] and in [28]. The model considered is specified by listing in the second and third column if it allows to employ, respectively, conflicts and capacity constraints. The fourth column reports

Model	Conflicts	Capacity constraints	Problem	Complexity
Aeolus^{-}	✗	✗	achievability	P
Aeolus core	✓	✗	reconfigurability	EXPSPACE
Aeolus flat	✓	✓	achievability	undecidable

Table 5.1: Results for the achievability problem.

the problem addressed: *reconfigurability* is a variant of the achievability problem where the deployment run potentially starts from a non-empty initial configuration. Notice that the problems considered are *decisional* in that the question they answer is if there exists a deployment (respectively a reconfiguration) run reaching a desired target component. The problem complexity varies a lot as it ranges from polynomial to undecidable! Notice that at the moment we are missing formal results for the case with capacity constraints and no conflicts.



(a) Aeolus flat

(b) Aeolus core

Figure 5.4: Components employed in the encodings.

Observation. It is worth mentioning that the proofs for establishing EXPSPACE complexity and undecidability are based on encoding the given problem into *Petri nets* [68] and *2 Counter Machines* [59], respectively. The Aeolus component types

employed in these encodings exhibit a highly complex behavior. This complexity is mirrored by means of automata with huge number of states and ports, as the ones depicted in Figure 5.4. It can be argued that real life scenarios are unlikely to demand such complex models. A possible approach built on this insight might try to fix the complexity of the automata allowed in the specification of a component type. One could, for instance, limit the number of states in the automaton definition. We will come back to this observation in Chapter 8 where we discuss future directions of research.

5.2 Other component models

Although Aeolus is not the first model employed to frame the problem of component deployment in a distributed setting, the combination of features it combines, represents an elegant formalization. Automata-representations enable to model complex life cycles of components. This coupled with the possibility to express capacity constraints for redundancy/load balancing requirements and, finally, a way to define conflicts among components result in a flexible and concise model, fit to represent the task of deploying complex configurations in a distributed setting. The simplicity of the model enabled to establish formal complexity results that shed light over the computational hardness of the deployment problem.

In the following several component models in the literature are recalled for comparison.

Automaton-based models have been adopted long ago in the context of component-oriented development frameworks. One of the most influential model is that of *interface automata* [26], where automata are used to represent the component behavior in terms of input, output, and internal actions. Interface automata support automatic compatibility check and refinement verification: a component refines another if its interface has weaker input assumptions and stronger output guarantees.

Differently from that approach, we are not interested in component compatibility or refinement, and we do not require complementary behavior of components: we

simply check in the current configuration whether all required functionalities are provided by currently deployed components. Moreover, the automata in Aeolus do not represent the internal behavior of components, but the effect on the component of an external deployment or reconfiguration actions.

Aeolus reconfiguration actions show interesting similarities with transitions in Petri nets [68], a very popular model born from the attempt to extend automata with concurrency. At first sight, one might encode our model in Petri net, representing our component states as places, each deployed component as a token in the corresponding place, and reconfiguration actions as transitions that cancel and produce tokens. Achievability in Aeolus would then correspond to *coverability* in Petri nets. But there are several important differences. Multiple state change actions can atomically change the state of an unbounded number of components, while in Petri net each transition consumes a predefined number of tokens. More importantly, we have proved that achievability can be solved in polynomial time for the Aeolus$^-$ fragment and that it is undecidable for the Aeolus flat model, while in Petri nets coverability is an ExpSpace problem [70].

Several process calculi extend/modify the π-calculus [72] in order to deal with software components. The Piccola calculus [14] extends the asynchronous π-calculus [72] with *forms*, first-class extensible namespaces, useful to model component interfaces and bindings. Calculi like KELL [73] and HOMER [22] extends a core π-calculus with hierarchical locations, local actions, higher-order communication, programmable membranes, and dynamic binding. More recently, MECo [61] has extended this approach by proposing also explicit component interfaces and channels to realize tunneling effects traversing the hierarchical location boundaries. All these proposals differ from Aeolus model because they focus on modeling component interactions and communication, while we focus on their interdependencies during system deployment and reconfiguration.

Another related model is the Fractal component model [21]. It focuses on expressivity and flexibility: it provides a general notion of component assembly that can be used to describe concisely, and independently of the programming language, a

complex software system. Building on Fractal, FraSCAti [74] provides a middleware that can be used to deploy applications in the cloud.

In this and the other models the goal is to allow the user to assemble a working system out of components that have been specifically designed or adapted to work together. Component selection and interconnection are the responsibility of the user, and if some reconfiguration needs to happen, it is either obtained by reassembling the system manually, or by writing specific code that is still the responsibility of the user.

While *expressivity* is certainly important, solving the cloud challenge also requires *automation*: when the number of components grows, or the need to reconfigure appears more frequently, it is essential to be able to specify at a certain level of abstraction a particular configuration of the distributed software system, and to develop tools that provide a set of possible evolution paths leading from the current system configuration to one that corresponds to a user request.

Part IV

Deployment planning

Overview

In this part is presented the actual contribution of the work developed during my PhD activity. The presentation is split in two chapters.

Chapter 6, is dedicated to present the ideas behind an ad-hoc planning technique, tailored to address the deployment problem. First, the specific variant of the Aeolus model adopted is described and the formal statement of the problem is provided. Then, a novel technique is presented in detail, together with formal results that guarantee the proposed algorithm to be sound, complete and efficient (its computational complexity being polynomial). The central results of this section are contained in [55].

The other, Chapter 7, surveys the development of a proof of concept planner, putting into practice the technique presented in previous chapter. Experimental results, used to validate the tool, are part of this chapter. This section relies on material partially presented in [53] (see also [54] for an extended version of previous paper).

Chapter 6

Theory

This chapter provides the formal account of the proposed approach.

Section 6.1 is dedicated to formally framing the problem, with the definition the Aeolus$^-$ fragment and of the *deployment problem*. Section 6.2 describes the technique developed to solve the latter. In order to ease the understanding of its technicalities we rely on a running example, showing the technique at work in a step by step way. Finally, Section 6.3 presents formal results that prove the correctness of the technique and its efficiency from a computational complexity point of view.

6.1 Aeolus$^-$ model & problem statement

In this section we introduce the fragment of the Aeolus model that we employ to frame the problem addressed.

The problem is the following: find an algorithm that, given a universe of components, computes a deployment plan, i.e. a correct sequence of actions leading to a configuration where a target component is in a given state.

In order to enable algorithms to efficiently compute a deployment plan, restrictions to the original model must be considered. Indeed, in the Aeolus model, a problem very similar to the deployment problem has been proven to be undecidable [30]! For a variant of the original model, called *Aeolus core*, where the possibility to specify conflicts is dropped, another version of the deployment problem has been proven

to be in the EXPSPACE complexity class, hence unfeasible in practice [28] (see [29] for an extended version).

The achievability problem, defined in Section 5.8, can be seen as the decisional variant of the deployment problem. In [30] a polynomial algorithm solving this problem has been devised, abstracting from the total number of instances of the same component and from individual bindings that form a configuration. Starting from this, developing an algorithm to solve the problem of actually computing a deployment plan has proven to be a non-trivial challenge. This is mainly due to the fact that one has to take into account the actual configurations with the bindings that are activated (and deactivated). For an interface required by a given component, there are possibly many different components that provide it. This means that different bindings are enabled. As different binding possibilities translate into disjunctions of logical conditions to be satisfied, it is hard to deal with the exponential explosion characterizing satisfiability problems.

We proceed with the formal definition of the restriction of the Aeolus component model, presented in Chapter 5.

Definition 6.1 (Component type)**.** *The set* $\mathcal{T}_{flat}$ *of* component types *ranged over by* $\mathcal{T}, \mathcal{T}_1, \mathcal{T}_2, \ldots$ *contains 4-ples* $\langle Q, q_0, T, D \rangle$ *where:*

- Q *is a finite set of states containing the initial state* q_0*;*
- $T \subseteq Q \times Q$ *is the set of* transitions*;*
- D *is a function from* Q *to a pair* $\langle \mathbf{P}, \mathbf{R} \rangle$ *of port names (i.e.* $\mathbf{P}, \mathbf{R} \subseteq \mathcal{I}$*) indicating the provided and required ports that each state activates. We assume that the initial state* q_0 *has no requirements (i.e.* $D(q_0) = \langle \mathbf{P}, \emptyset \rangle$*).*

Notice that function D has been modified: it only associates to every state a pair of required and provided interfaces. By changing its range we rule out at the same time the possibility to specify capacity constraints and conflicts (as the former are encoded as a capacity constraint with 0 value).

We now define configurations that describe systems composed by components and their bindings. Each component has a unique identifier, taken from the set $\mathcal{Z}$. A configuration, ranged over by $\mathcal{C}_1, \mathcal{C}_2, \ldots$, is given by a set of component types, a set of components in some state, and a set of bindings.

Definition 6.2 (Configuration). *A* configuration $\mathcal{C}$ *is a 4-ple* $\langle U, Z, S, B \rangle$ *where:*

- $U \subseteq \mathcal{T}_{flat}$ *is the finite* universe *of the available component types;*
- $Z \subseteq \mathcal{Z}$ *is the set of the currently deployed* components*;*
- S *is the component* state description*, i.e. a function that associates to components in* Z *a pair* $\langle \mathcal{T}, q \rangle$ *where* $\mathcal{T} \in U$ *is a component type* $\langle Q, q_0, T, D \rangle$*, and* $q \in Q$ *is the current component state;*
- $B \subseteq \mathcal{I} \times Z \times Z$ *is the set of* bindings*, namely 3-ple composed by a port, the component that provides that port, and the component that requires it; we assume that the two components are distinct.*

Notation. We write $\mathcal{C}[z]$ as a lookup operation that retrieves the pair $\langle \mathcal{T}, q \rangle = S(z)$, where $\mathcal{C} = \langle U, Z, S, B \rangle$. On such a pair we then use the postfix projection operators `.type` and `.state` to retrieve $\mathcal{T}$ and q, respectively. Similarly, given a component type $\langle Q, q_0, T, D \rangle$, we use projections to decompose it: `.states`, `.init`, and `.trans` return the first three elements; $.\mathbf{P}(q)$ and $.\mathbf{R}(q)$ return the two elements of the $D(q)$ tuple. Moreover, we use `.prov` (resp. `.req`) to denote the union of all the provided ports (resp. required ports) of the states in Q. When there is no ambiguity we take the liberty to apply the component type projections to $\langle \mathcal{T}, q \rangle$ pairs. *Example*: $\mathcal{C}[z].\mathbf{R}(q)$ stands for the required ports of component z in configuration $\mathcal{C}$ when it is in state q.

A configuration is correct if all the active required ports are bound to provided ports that are active.

Definition 6.3 (Correctness). *Let us consider a configuration* $\mathcal{C} = \langle U, Z, S, B \rangle$.

We write $\mathcal{C} \models_{req} (z, r)$ *to indicate that the required port of component* z*, with interface* r*, is bound to an active port providing* r*, i.e. there exists a component* $z' \in Z \setminus \{z\}$ *such that* $\langle r, z', z \rangle \in B$*,* $\mathcal{C}[z'] = \langle \mathcal{T}', q' \rangle$ *and* r *is in* $\mathcal{T}'.\mathbf{P}(q')$*.*

The configuration $\mathcal{C}$ is correct *if for every component $z \in Z$ with $S(z) = \langle \mathcal{T}, q \rangle$ we have that $\mathcal{C} \models_{req} (z, r)$ for every $r \in \mathcal{T}.\mathbf{R}(q)$.*

We now formalize how configurations evolve by means of actions.

Definition 6.4 (Actions)**.** *The set $\mathcal{A}$ contains the following actions:*

- *$stateChange(z, q, q')$ changes the state of the component $z \in \mathcal{Z}$ from q to q'*
- *$bind(r, z_1, z_2)$ creates a binding between the provided port $r \in \mathcal{I}$ of the component z_1 and the required port r of z_2 ($z_1, z_2 \in \mathcal{Z}$);*
- *$unbind(r, z_1, z_2)$ deletes the binding between the provided port $r \in \mathcal{I}$ of the component z_1 and the required port r of z_2 ($z_1, z_2 \in \mathcal{Z}$);*
- *$new(z : \mathcal{T})$ creates a new component of type $\mathcal{T}$ in its initial state. The new component is identified by a unique and fresh identifier $z \in \mathcal{Z}$;*
- *$del(z)$ deletes the component $z \in \mathcal{Z}$.*

The execution of actions is formalized by means of a labeled transition system on configurations, which uses actions as labels.

Definition 6.5 (Reconfigurations)**.** *Reconfigurations are denoted by transitions $\mathcal{C} \xrightarrow{\alpha} \mathcal{C}'$ meaning that the execution of $\alpha \in \mathcal{A}$ on the configuration $\mathcal{C}$ produces a new configuration $\mathcal{C}'$. The transitions from a configuration $\mathcal{C} = \langle U, Z, S, B \rangle$ are defined as follows:*

$$\mathcal{C} \xrightarrow{stateChange(z,q,q')} \langle U, Z, S', B \rangle$$
$$\text{if } \mathcal{C}[z].\texttt{state} = q \text{ and } (q, q') \in \mathcal{C}[z].\texttt{trans} \text{ and } S'(z') = \begin{cases} \langle \mathcal{C}[z].\texttt{type}, q' \rangle & \text{if } z' = z \\ \mathcal{C}[z'] & \text{otherwise} \end{cases}$$

$$\mathcal{C} \xrightarrow{bind(r,z_1,z_2)} \langle U, Z, S, B \cup \langle r, z_1, z_2 \rangle \rangle$$
$$\text{if } \langle r, z_1, z_2 \rangle \notin B \text{ and } r \in \mathcal{C}[z_1].\texttt{prov} \cap \mathcal{C}[z_2].\texttt{req}$$

$$\mathcal{C} \xrightarrow{unbind(r,z_1,z_2)} \langle U, Z, S, B \setminus \langle r, z_1, z_2 \rangle \rangle \quad \text{if } \langle r, z_1, z_2 \rangle \in B$$

$$\mathcal{C} \xrightarrow{new(z:\mathcal{T})} \langle U, Z \cup \{z\}, S', B \rangle$$
$$\text{if } z \notin Z,\ \mathcal{T} \in U \text{ and } S'(z') = \begin{cases} \langle \mathcal{T}, \mathcal{T}.\texttt{init} \rangle & \text{if } z' = z \\ \mathcal{C}[z'] & \text{otherwise} \end{cases}$$

$$\mathcal{C} \xrightarrow{del(z)} \langle U, Z \setminus \{z\}, S', B' \rangle$$
$$\text{if } S'(z') = \begin{cases} \bot & \text{if } z' = z \\ \mathcal{C}[z'] & \text{otherwise} \end{cases} \text{ and } B' = \{\langle r, z_1, z_2 \rangle \in B \mid z \notin \{z_1, z_2\}\}$$

We can now define a *deployment plan* as a sequence of actions that transform a correct configuration (not necessarily initial) without violating correctness along the way.

Definition 6.6 (Deployment plan)**.** *A* deployment plan P *is a sequence of reconfigurations* $\mathcal{C}_0 \xrightarrow{\alpha_1} \mathcal{C}_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_m} \mathcal{C}_m$ *such that* $\mathcal{C}_i$ *is correct, for* $0 \leq i \leq m$*.*

We can now formulate the problem addressed, i.e. the *deployment problem*. Given an universe of component types, we want to know whether it is possible to deploy at least one component of a given component type $\mathcal{T}$ in a given state q. Moreover, we want to effectively synthesize a deployment plan, specifying a sequence of steps that enable one to deploy the target component.

Definition 6.7 (Deployment problem)**.** *The* deployment problem *has as input an universe* U *of component types, a target component type* $\mathcal{T}_t$*, and a target state* q_t*. The output is a deployment plan* $\mathsf{P} = \mathcal{C}_0 \xrightarrow{\alpha_1} \mathcal{C}_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_m} \mathcal{C}_m$ *such that* $\mathcal{C}_0 = \langle U, \emptyset, \emptyset, \emptyset \rangle$ *and* $\mathcal{C}_m[z] = \langle \mathcal{T}_t, q_t \rangle$*, for some component* z *in* $\mathcal{C}_m$*, if there exists one. Otherwise, it returns a negative answer, stating that no such a plan exists.*

Remark 6.1. *As already mentioned in Remark 5.1, the limiting assumption to seek for a single component does not hinder the generality of the problem. Encoding a full configuration* $\mathcal{C}$ *by adding a dummy target* $\langle \mathcal{T}, q \rangle$ *whose state* q *requires (dummy) ports provided by the final states of components in* $\mathcal{C}$*. The modifications that need to be performed on the original desired configuration* $\mathcal{C}$ *in order to deal with the encoding are limited:*

- *every final state* s_i^f *of each component* $\mathcal{T}_i$ *in* $\mathcal{C}$ *must enable an additional (fictitious) provided port* p_i^f*;*
- *add a new component with 2 states:* stop *and* start *s.t. the latter has one required port* p_i^f *per additional provided one.*

After applying this change one performs the usual deployment plan synthesis, as explained below. The output will be a plan P' *to reach the modified final configuration* $\mathcal{C}^+$*. A post-processing phase could then remove all the steps involving the*

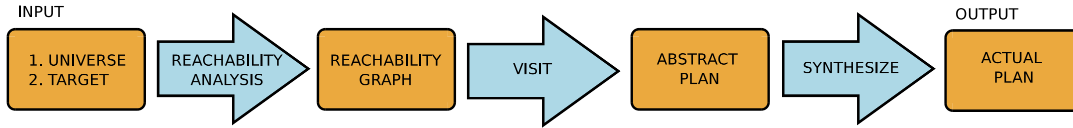


Figure 6.1: Chain of three phases.

additional dummy component $\mathcal{T}$*, thus obtaining the plan* P *for the originally desired configuration* $\mathcal{C}$*.*

Next chapter presents the proposed solution to the above problem, representing the original contribution of this work.

6.2 Technique

The technique developed to tackle the deployment problem consists in an algorithm that is a chain of three phases: *reachability analysis*, *abstract planning* and *plan synthesis*.

As depicted by Figure 6.1 each phase works on an intermediate representation output by the previous one. The input to the algorithm, according to Definition 6.7, is a universe U of component types and a target $\langle \mathcal{T}_t, q_t \rangle$, that represents the target state q_t of the (target) component type $\mathcal{T}_t$. By performing the first step, *reachability analysis*, a data structure called *reachability graph* is built. This representation bears information on the component types that the deployment plan will employ. If the target is not reachable, the algorithm raises an exception stating that no solution exists for the problem, otherwise the algorithm proceeds with the subsequent phase, called *abstract planning*. This phase basically corresponds to a bottom-up visit of the reachability graph. Its aim is to select the component type-state pairs to be used and establish the necessary bindings between the activated provided and required ports. At the end of this phase, an intermediate representation is generated, named *abstract plan*. This representation is a graph where nodes represent deployment actions and arcs denote temporal/logical dependencies among them. It is *abstract*

in that for each component type there is only one representative instance. Finally, the *plan synthesis* phase uses information from the abstract plan to produce the actual deployment plan where concrete instances appear.

The pseudocode of the technique can be simply summarized as a sequence of steps described by the following algorithm:

Algorithm 1 *DeploymentPlanner* pseudocode

1: perform reachability analysis
2: **if** target is not generated **then**
3: raise exception `TargetNotReachable` and abort
4: **else**
5: perform component selection
6: generate abstract plan
7: synthesize plan

Running example Consider the task of setting up a MySQL *master-slave replication*, involving two databases. This typical MySQL configuration is used in solutions that facilitate data backup, analyzing data without using the main database or as a means to scale-out. Essentially the process consists in copying the data from the primary database, called the *master*, to the secondary one, called the *slave* and authorizing the latter to read the log of the master. As *read* operations can be performed on any of the slave nodes and *master* can be fully dedicated to *write* operations, a general performance improvement of the system is achieved.
In order to set up this configuration several steps need to be carried out. Initially, the master node must be installed, configured and put in running mode to start serving external requests. Then, the slave has to be activated. This involves a kind of two-steps protocol in which: first, the slave authenticates itself to the master and then the latter can send to the slave a *dump*, i.e. a snapshot of its data. This means that there is a circular dependency between master and slave, since the latter requires the dump of the former that, on its turn, requires the IP address of the

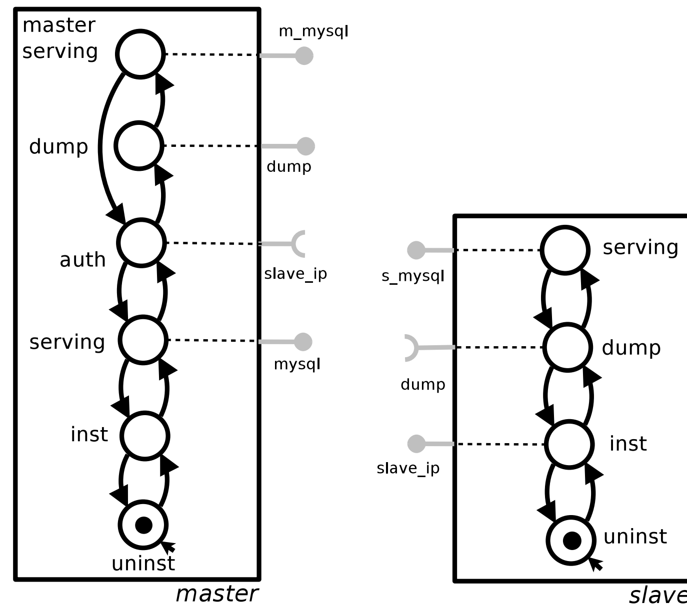


Figure 6.2: MySQL master-slave components according to the Aeolus model

slave in order to grant it replication privileges [1].

By relying on the Aeolus⁻ model, it is possible to describe the master and slave by means of the component types depicted in Figure 6.2. The master component has six states, an initial uninst state followed by inst and serving. In serving state, it activates port *mysql*. When the master-slave replication configuration is needed, in order to enter the final master serving state, it first traverses state auth that requires the IP address of the slave, and state dump to provide the dump to the slave. State master serving provides port *m_mysql* which describes the fact that an additional database, acting as slave, has been set up. The slave component is instead described by an automaton with four states. The initial uninst state is followed by state inst which provides the IP address by means of the provided *slave_ip* port. Subsequent state dump requires the dump from the master by means of homonymous port *dump*.

[1]Notice that it is possible to grant permission to all slaves at once, thus breaking the circular dependency. However, for security reasons, usually authorization is granted independently to each slave individually via command `GRANT REPLICATION SLAVE ON *.* TO 'slave-user'@'slave-address' IDENTIFIED BY 'password';`.

Finally, slave can reach state serving, that provides interface *s_mysql*, required by the target component.

The desired final configuration can be easily specified by using an additional component that requires both ports *m_mysql* and *s_mysql*, provided respectively by state master serving of component master and by state serving of component slave. Figure 6.3 depicts this additional target component, called application, that in its inst state requires the presence of both a master in state master serving and a slave in state serving.

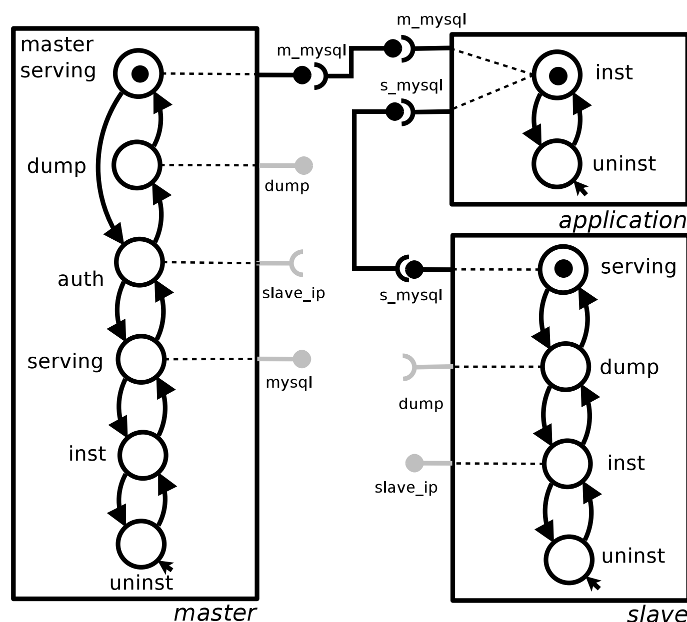
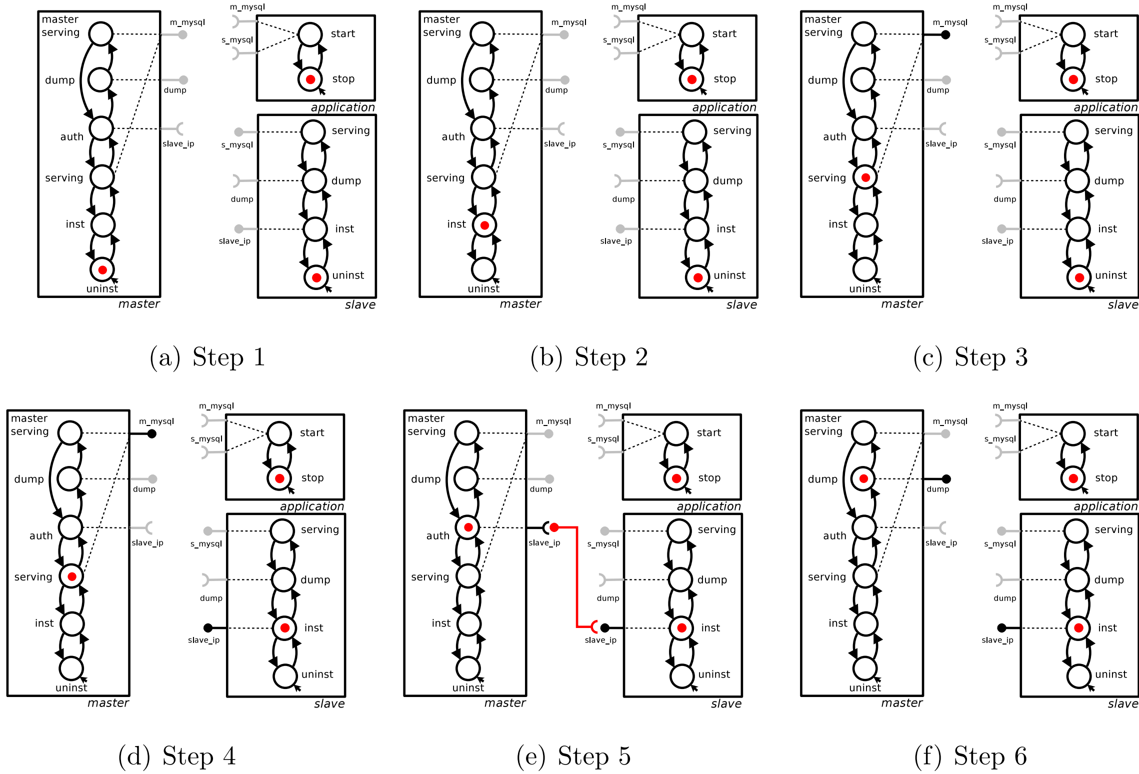


Figure 6.3: MySQL master-slave replication final configuration

As an example of a deployment plan let us consider the configuration depicted in Figure 6.2. If we want to activate the slave, a possible deployment plan that allows to achieve this, requires to perform two consecutive *stateChange* actions in the master to reach the dump state. At this point, the slave component can reach the serving state performing first the state change into the dump state and then into the serving state. Figure 6.4 and Figure 6.5 depict graphically the steps involved in the deployment plan sketched above. Note that every action in the deployment plan

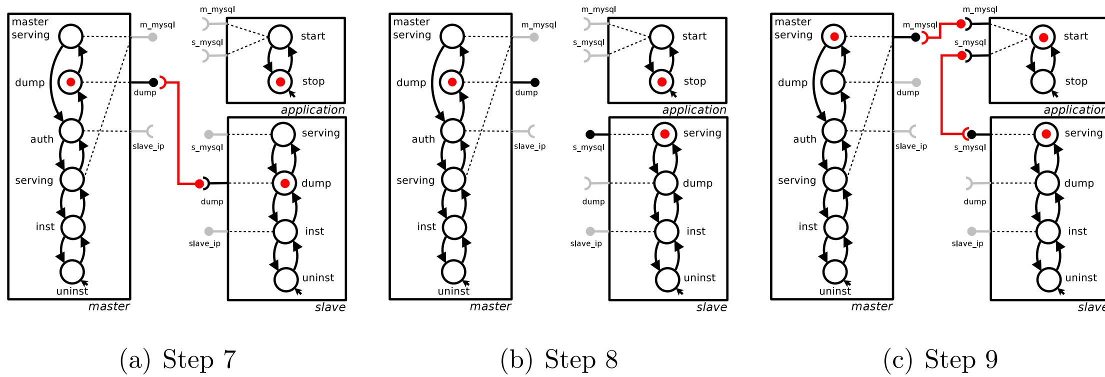
will correspond to one or more concrete instructions. For instance, the state change from the serving to the auth state in the master corresponds to issue the command `GRANT REPLICATION SLAVE ON *.* TO 'slave-user'@'slave-ip';`.



(a) Step 1 (b) Step 2 (c) Step 3

(d) Step 4 (e) Step 5 (f) Step 6

Figure 6.4: Sample deployment plan for the running example (part 1).



(a) Step 7 (b) Step 8 (c) Step 9

Figure 6.5: Sample deployment plan for the running example (part 2).

6.2.1 Reachability analysis

The aim of the first phase is to check if the target can be obtained starting from an initial empty configuration. This is achieved through a forward symbolic reachability analysis that relies on an abstract representation of components. For each component its individual identity as well as the number of its instances are abstracted away, keeping only its component type and its state $\langle \mathcal{T}, q \rangle$. Also, we ignore individual bindings and avoid considering *del* actions. The abstraction from the bindings is allowed since one can safely assume that, given a set of components, all complementary ports on two distinct components are bound. [2] Delete actions are superfluous since the presence of one component does not hinder the reachability of a state in another component.

The algorithm works by *saturation*, producing iteratively new generations of component type-state pairs that become available as soon as they are reachable from nodes in current generation and all their requirements can be fulfilled by already existing pairs. This process is repeated until a fix-point is reached, i.e. no new pairs are added. The soundness of this "forward approach" comes from the fact that in the Aeolus$^-$ model the set of available nodes is monotonically increasing, i.e. a valid

[2]Remember that we are not taking into account capacity constraints. This means that a provided port may be connected to an unbounded number of components requiring it.

configuration will stay valid if we add new nodes [3].

Algorithm 2 Reachability graph construction

1: $Nodes_0 = \{\langle \mathcal{T}, \mathcal{T}.\mathtt{init} \rangle \mid \mathcal{T} \in U\}$; $provPort = \bigcup_{\langle \mathcal{T}, q \rangle \in Nodes_0} \{\mathcal{T}.\mathbf{P}(q)\}$; $i = 0$;
2: **repeat**
3: $i = i + 1$;
4: $Arcs_i, Nodes_i = \emptyset$;
5: **for all** $\langle \mathcal{T}, q \rangle \in Nodes_{i-1}$ **do**
6: **for all** $(q, q') \in \mathcal{T}.\mathtt{trans}$ **do**
7: **if** $\mathcal{T}.\mathbf{R}(q') \subseteq provPort$ **then**
8: $Nodes_i.\mathtt{add}(\langle \mathcal{T}, q' \rangle)$;
9: **for all** $\langle \mathcal{T}, q \rangle \in Nodes_i$ **do**
10: $provPort.\mathtt{add}(\mathcal{T}.\mathbf{P}(q))$;
11: $Nodes_i = Nodes_{i-1} \cup Nodes_i$
12: **for all** $\langle \mathcal{T}, q \rangle \in Nodes_{i-1}, \langle \mathcal{T}, q' \rangle \in Nodes_i$ **do**
13: **if** $(q, q') \in \mathcal{T}.\mathtt{trans}$ **then**
14: $Arcs_i.\mathtt{add}(\langle \mathcal{T}, q' \rangle \longrightarrow \langle \mathcal{T}, q \rangle)$;
15: **if** $q == q'$ **then**
16: $Arcs_i.\mathtt{add}(\langle \mathcal{T}, q' \rangle \cdots\cdots \langle \mathcal{T}, q \rangle)$;
17: **until** $Nodes_{i-1} == Nodes_i$

The first phase outputs a data structure, called *reachability graph*, that looks like a pyramid where the top level contains all the component types in their initial state and, at every step, a new level is produced by adding new component type-state pairs, reachable from the ones at current level by means of *stateChange* actions.

Figure 6.6 depicts the final reachability graph for the MySQL master-slave replication case study. The target pair is highlighted in red. The first level of Figure 6.6 contains components M, S and A in their initial state. At the second level, two

[3] Notice that this is true as long as one banishes the possibility to specify conflicts. In presence of conflicts the introduction of a component in a given state may invalidate correctness of the current configuration.

pairs are added: component M in I and component S in I, derived respectively from M in U and S in U. At level 3, pair $\langle \mathcal{M}, S \rangle$ is added. At next step, pair $\langle \mathcal{M}, A \rangle$ can also be added since it derives from $\langle \mathcal{M}, S \rangle$ and its requirement on the interface *slave_ip* is fulfilled by $\langle \mathcal{S}, I \rangle$, appearing at previous level. Now, $\langle \mathcal{M}, D \rangle$ becomes reachable as it can be derived from $\langle \mathcal{M}, A \rangle$. Two new pairs appear at next level, namely $\langle \mathcal{M}, RS \rangle$ and $\langle \mathcal{S}, D \rangle$. The latter is derived from $\langle \mathcal{S}, I \rangle$ and his *dump* require is fulfilled by $\langle \mathcal{M}, D \rangle$ at previous level, whereas the former has no requirements. At level 7, $\langle \mathcal{S}, S \rangle$ is added as it is derivable from $\langle \mathcal{S}, D \rangle$. Finally, the target node, pair $\langle \mathcal{A}, I \rangle$ is added when both its requirements, *m_mysql* and *s_mysql* are provided by pairs in previous level, namely $\langle \mathcal{M}, RS \rangle$ and $\langle \mathcal{S}, S \rangle$. This is the last level as no new type-state pairs can be generated.

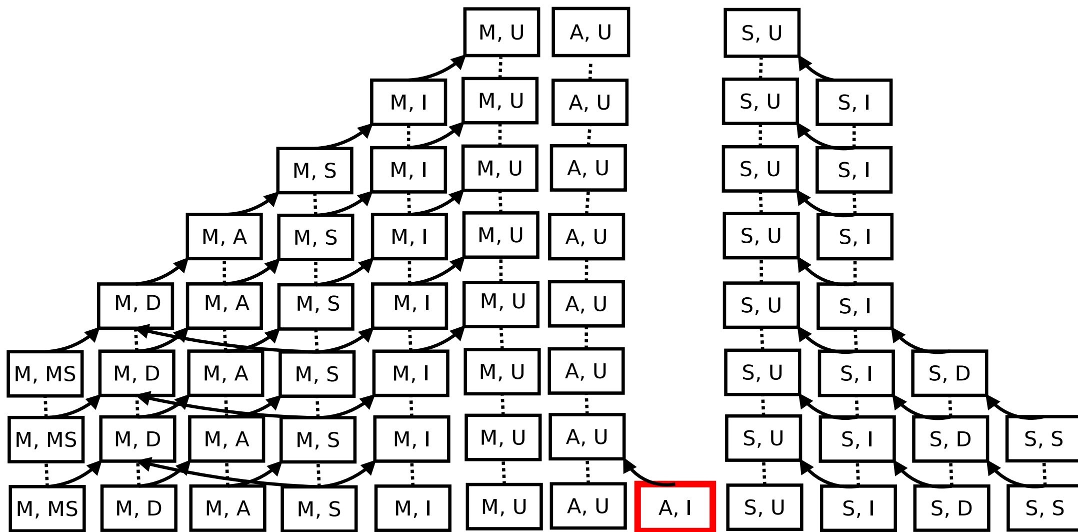


Figure 6.6: Reachability graph for the running example.

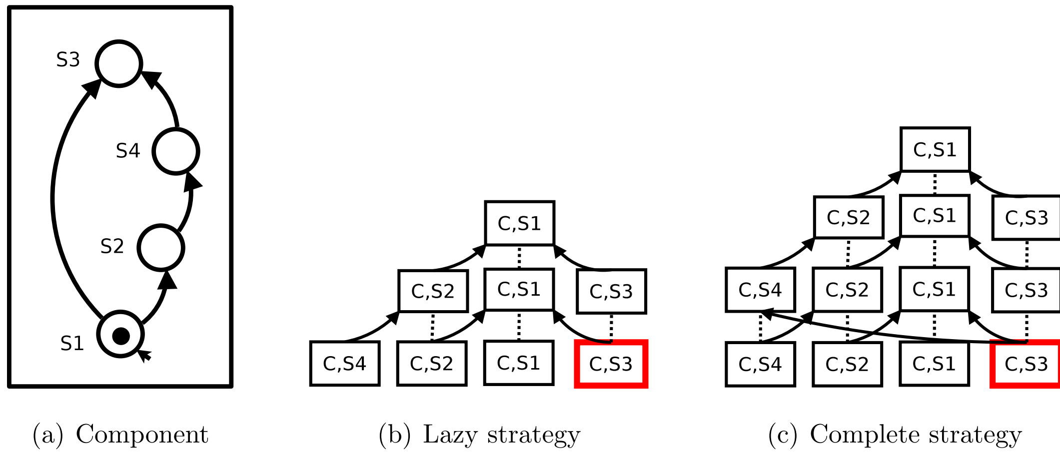
Algorithm 2 performs the reachability graph construction. Variable $Nodes_i$ is used to denote the set of the type-state pairs at level i, while $Arcs_i$ represents all the possible ways a type-state pair can be obtained. There are two kinds of arcs: $x \longrightarrow y$ means that component type-state pair y, at level $i+1$, can be obtained from x at level i by means of a *stateChange* action; $x \cdots\cdots y$ means y is a copy of an already existing pair x. Finally, $ProvPort$ is used to store the ports provided

by the pairs currently provided. [4] Initially, it contains the ports provided by all components in their initial state (line 1) and then it is incrementally augmented with the ports provided by the newly added components (lines 9-10). The new type-state pairs to be added are computed by checking if all their requirements are satisfied by at least one component state at the previous level (lines 5-8). Finally, variable $Arcs_i$ is updated (lines 13-16), listing all the possible ways a type-state pair can be obtained. The generation of levels proceeds until a fix-point is reached (line 17). Termination is guaranteed by the fact that the number of possible type-state pairs is finite and at every iteration at least a new pair is added to the $Node_i$ set. Once the fix-point has been reached, if the bottom-level generation does not contain the target component type-state pair, a plan to achieve the goal does not exist and we can interrupt execution of the subsequent phases of the algorithm.

Lazy & complete strategy Let us denote with k the total number of different type-state pairs. In the worst case Algorithm 2 adds at every iteration only one pair and so the upper bound on the number of iterations is k. There are a few possible strategies to decide the level at which to stop the iteration for building up the reachability graph. One could employ, for instance, a *lazy strategy*, stopping as soon as the target node is produced, if ever. Another possibility is a *complete strategy* that enforces to go on anyway, until the fix-point is reached. The lazy strategy minimizes the work to be done in this phase but has the drawback that not all paths to reach a certain pair may have been discovered yet. Figure 6.7 shows a simple example highlighting the difference between the reachability graph obtained applying a lazy and the complete strategy. Consider as target component C in state S3. The reachability graph, built using the complete strategy, has one layer more where a new path S4 $\longrightarrow$ S3 to the target is discovered by an additional iteration of the algorithm. In the general case an alternative path may present advantages over the others and so the lazy strategy may result in a poorer deployment plan as there

[4] For simplicity we employ a single set for this purpose as the set of provided ports is monotonically increasing.

may be less choices available.



(a) Component (b) Lazy strategy (c) Complete strategy

Figure 6.7: Difference between lazy and complete strategies.

6.2.2 Abstract planning

If the target state is reachable, the *abstract planning* phase generates a different graph-like representation that indicates the necessary state change actions and the causal dependencies among them. Causal dependencies reflect, for instance, the fact that a component should enter a state enacting a provide port before another component enters a state requiring that port. This information is captured by the so-called *abstract plan*, output by this phase. The abstract plan specifies the life-cycle of the component types that will be employed in the deployment of the target state.

The first step in generating the abstract plan is to identify the components to be used in the deployment plan. A state may have multiple predecessors in the automaton description of a component type's behaviour. Hence, in the reachability graph, a pair p may have multiple origin pairs: it simply means p is reached by means of a *stateChange* action performed on any of its predecessors. Moreover, a pair becomes reachable if the required ports of its state can be bound to ports provided by pairs in previous generation. In general there may be more than one

node providing the needed port. Defining a deployment plan involves specifying for each component employed how it is reached: which node gives origin to it and which nodes are the providers for its requirements. The alternative possibilities in both cases, imply a choice must be made for each node. The first step in this phase deals precisely with these aspect. Starting from the target pair at the bottom of the reachability graph, a selection procedure is carried out in order to pick the pairs to be employed.

Figure 6.8 depicts a possible component selection for the MySQL master-slave example. Selected nodes are highlighted in red. For space reasons, master, slave and application are denoted by their initials M, S and A respectively, and each state is referred by its initial upper-case letter: U for uninst, I for inst, S for serving, A for auth, D for dump, S for serving and MS for master serving.

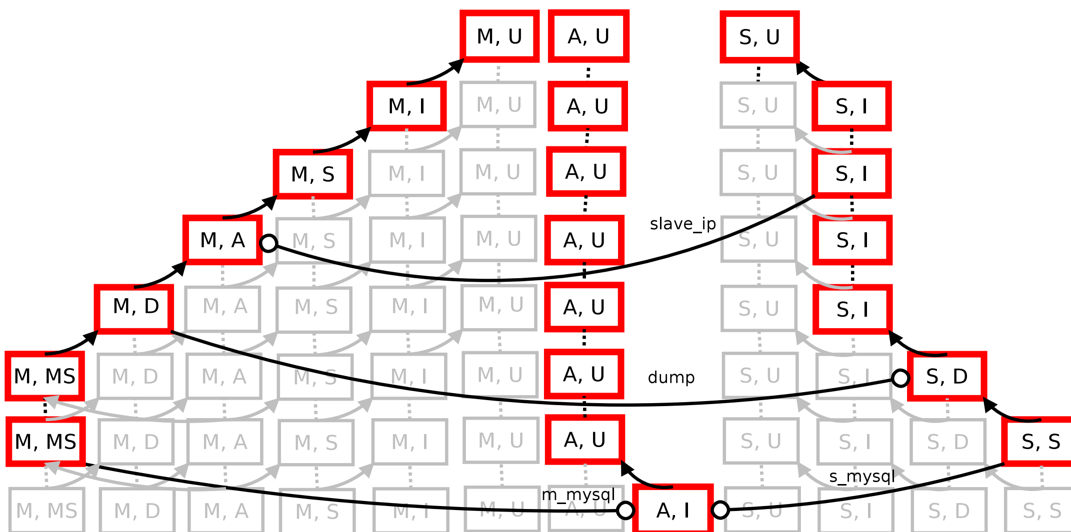


Figure 6.8: Component selection for the running example.

The selection procedure basically amounts to find a path to the root level for every pair that is selected. The target node, $\langle \mathcal{A}, I \rangle$ in the last level is the starting point. There is only one possible derivation for $\langle \mathcal{A}, I \rangle$ and so $\langle \mathcal{A}, U \rangle$ is selected as its origin. Since $\langle \mathcal{A}, I \rangle$ requires two interfaces, *m_mysql* and *s_mysql*, provided by $\langle \mathcal{M}, MS \rangle$ and $\langle \mathcal{S}, S \rangle$, these nodes, that will be acting as providers are also selected.

The selection process continues until the root level is reached.

Component selection is performed by means of a bottom-up visit of the reachability graph, described by Algorithm 3. From the bottom level, denoted by n, it proceeds upward selecting the pairs used to deploy each selected pair appearing at the current level.

Algorithm 3 Component Selection

```
1: SNodes_n = {⟨T_target, q_target⟩};
2: for i = n downto 1 do
3:     SNodes_{i−1} = SArcs_{i−1} = ∅;
4:     for all ⟨T, q⟩ ∈ SNodes_i do
5:         ⟨T′, q′⟩ = heuristic_parent(⟨T, q⟩, i);
6:         SNodes_{i−1}.add(⟨T′, q′⟩);
7:         SArcs_{i−1}.add(⟨T, q⟩ → ⟨T′, q′⟩);
8:         for all r ∈ T.R(q) do
9:             ⟨T′, q′⟩ = heuristic_prov(⟨T, q⟩, r, i);
10:            SNodes_{i−1}.add(⟨T′, q′⟩);
11:            SReq.add(⟨T′, q′⟩ −r−∘ ⟨T, q⟩);
```

Variables $SNodes_i$ and $SArcs_i$ are used to keep track of the selected component-state pairs at level i and how these pairs are obtained. From the last level only the target pair is selected (line 1). For every selected component $\langle \mathcal{T}, q \rangle$ at level $i+1$, we select at level i one of its predecessors: $\langle \mathcal{T}', q' \rangle$ becomes the origin of the given component. Consequently $\langle \mathcal{T}', q' \rangle$ is added to $SNodes_{i-1}$ and an $\langle \mathcal{T}, q \rangle \rightarrow \langle \mathcal{T}', q' \rangle$ arc is added to $SArcs_{i-1}$ (lines 5-7). As already mentioned, there may be more valid choices as a state may have more than one predecessor. For the choice of the origin node we rely on heuristics, here abstracted by function `heuristic_parent`. Discussions on the employed heuristics is deferred to Section 6.2.4.

For every required port needed by the selected pairs of level $i+1$ that are not copies, we select a pair at level i that is able to activate a complementary provided port. This choice is recorded in $SNodes_{i-1}$ and $SReq$ (lines 10-11). In

particular, variable $SReq$ is used to keep track of the bindings, between provided and required ports of components, that will be built during the deployment. This kind of dependency is represented by an $\langle \mathcal{T}', q' \rangle \stackrel{r}{\multimap} \langle \mathcal{T}, q \rangle$ arc where $\langle \mathcal{T}', q' \rangle$ is the component type-state pair that activates the provided port r, while $\langle \mathcal{T}, q \rangle$ activates the complementary required port. As for the choice of the origin node, we rely on heuristics, dubbed `heuristic_prov`, to decide which pair is used as a provider for the required ports.

Heuristics Both of these choices, the origin choice and the providers' choice, affect the length of the deployment plan. The main goal that we seek is to generate a deployment plan involving the least amount of components. As for both origin and provider there are potentially multiple allowed choices the goal corresponds roughly to find a global minimum in a setting with disjunctions of conditions. This is a typical NPO problem [5] and finding an exact solution to it is unfeasible in practice. In order to deal with this issue we have to rely on heuristics. We expand this topic in Section 6.2.4.

Once the component selection procedure is completed, the second phase proceeds to build the abstract plan. This representation can be seen as a directed graph where nodes represent either a *new*, *del*, or *stateChange* action, and arcs represent temporal precedence constraints. Each row represents the life-cycle of an instance of a given component type. Every node is tagged by a triple denoting an action in the following way:

- $\langle z, q, q' \rangle$ for a *stateChange* from state q to q' of instance z;
- $\langle z, \varepsilon, q_0 \rangle$ for a *new* action of instance z;
- $\langle z, q, \varepsilon \rangle$ for *del* action on instance z.

Precedence arcs are of three kinds, summarized in Table 6.1.

[5] A NP-Optimization (NPO) problem is a kind of Optimization problem whose corresponding decision problem is in NP. For a detailed characterization of NPO problems refer to [47].

Arc type	Meaning
$\longrightarrow$	precedence of *stateChange* actions on the same instance line. Formally $\langle z, x, x' \rangle \longrightarrow \langle z, x', x'' \rangle$ where x' is a state and x, x'' are either states or the special symbol ε denoting deletion of instance z.
$\overset{r}{\twoheadrightarrow}$	Precedence of instances that provide a port r w.r.t instances requiring it. Formally, $\langle z, x, y \rangle \overset{r}{\twoheadrightarrow} \langle z', x', y' \rangle$ if component z' in state y' requires r which is provided by z in state y (then state y must be entered before entering state y').
$\overset{r}{\dashrightarrow}$	Precedence of an instance requiring a port r w.r.t. actions that deactivate it. Formally, $\langle z, x, y \rangle \overset{r}{\dashrightarrow} \langle z', x', y' \rangle$ if component z' in state y' stops providing interface r which is required by z in state x (then state x must be left before z' enters state y').

Table 6.1: Kind of temporal precedence arcs.

Figure 7.1 displays the abstract plan for the running example. The rows represent the life-cycles of master, slave and application, respectively. The $\overset{slave_ip}{\twoheadrightarrow}$ from $\langle s, U, I \rangle$ to $\langle m, S, A \rangle$ expresses the fact that the *stateChange* of slave from uninstalled to installed must precede the *stateChange* of master from serving to auth because state auth of server requires interface *slave_ip*, provided by slave in state installed. The twin $\overset{slave_ip}{\dashrightarrow}$ arc states that master must switch from auth to dump before slave switches from installed to dump, as this state ceases providing interface *slave_ip*, otherwise its requirement would become unfulfilled. Following the same principle we can interpret the pair of arcs $\langle m, A, D \rangle \overset{dump}{\twoheadrightarrow} \langle s, I, D \rangle$ and $\langle s, D, S \rangle \overset{dump}{\dashrightarrow} \langle m, D, MS \rangle$ for interface *dump*. Finally, the target is represented by node $\langle a, U, I \rangle$, namely application entering state installed. This state requires two interfaces, *m_mysql* and *s_mysql* provided respectively by master in state master serving and slave in state

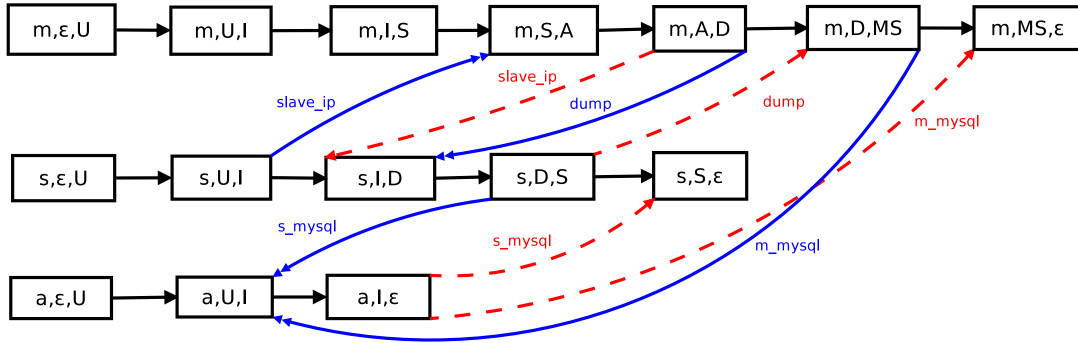


Figure 6.9: Abstract plan for the running example.

serving. Two $\twoheadrightarrow$ arcs (together with their $\dashrightarrow$ counterparts) are thus added with destination $\langle a, U, I\rangle$, one from $\langle s, D, S\rangle$ and the other one from $\langle m, D, MS\rangle$.

The procedure for the abstract plan generation is described by Algorithm 4. To generate an abstract plan we consider an instance for every maximal path in the previous representation, that starts from a type-state pair in the top level and reaches a type-state that is not a copy. Figure 6.8 shows, for instance, that there are three maximal paths, for the running example: one for the master (starting from $\langle \mathcal{M}, U\rangle$ and ending in $\langle \mathcal{M}, MS\rangle$), one for the application component and one for the slave (starting from $\langle \mathcal{S}, U\rangle$ and ending in $\langle \mathcal{S}, S\rangle$). The computation of the maximal paths is performed by invoking function `getMaxPaths` (line 1). [6] Every vertex in the abstract plan corresponds to an action. Variables Act and $Prec$ are used to store, respectively, the visited nodes/actions and the precedence constraints among them. The first loop (lines 3-12) is used to generate the nodes of the abstract plan and the precedence constraints $\longrightarrow$ among them. First of all, a new fresh name for each instance is generated (line 4) and is associated to the component type of the instance using map $InstMap$ (line 5). After that, nodes corresponding to the creation and deletion of the instance are added (line 6), as well as nodes representing intermediate state changes (line 8). The last part of the loop (lines 9-12) is used to generate the precedence arcs $\longrightarrow$. The second loop, starting at line 13, adds for

[6] We consider function `getMaxPaths` as given as it is easy to code and it does not give any particular insight.

Algorithm 4 Abstract Plan Generation

1: $Paths = \texttt{getMaxPaths}(Nodes_0, \ldots, Nodes_n)$;
2: $Act = \emptyset$; $InstMap = \{\,\}$;
3: **for all** $\big(\langle \mathcal{T}, q_0\rangle, \ldots, \langle \mathcal{T}, q_h\rangle\big) \in Paths$ **do**
4: $\quad inst = \texttt{getFreshName}()$;
5: $\quad InstMap[inst] = \mathcal{T}$;
6: $\quad Act.\texttt{add}(\langle inst, \varepsilon, q_0\rangle)$; $Act.\texttt{add}(\langle inst, q_h, \varepsilon\rangle)$;
7: $\quad$ **for all** $i \in [0..h-1]$ **do**
8: $\quad\quad Act.\texttt{add}(\langle inst, q_i, q_{i+1}\rangle)$
9: $\quad Prec.\texttt{add}(\langle\langle inst, \varepsilon, q_0\rangle \longrightarrow \langle inst, q_0, q_1\rangle\rangle)$;
10: $\quad Prec.\texttt{add}(\langle\langle inst, q_{h-1} q_h\rangle \longrightarrow \langle inst, q_h, \varepsilon\rangle\rangle)$;
11: $\quad$ **for all** $i \in [0..h-2]$ **do**
12: $\quad\quad Prec.\texttt{add}(\langle\langle inst, q_i, q_{i+1}\rangle \longrightarrow \langle inst, q_{i+1}, q_{i+2}\rangle\rangle)$;
13: **for all** $\langle\langle \mathcal{T}, q'\rangle \overset{r}{\multimap} \langle \mathcal{T}', s'\rangle\rangle \in SReq$ **do**
14: $\quad$ **for all** $n_1 == \langle i_1, s, s'\rangle \in Act$. $InstMap[i_1] == \mathcal{T}'$ **do**
15: $\quad\quad$ **let** $n_2 = \langle i_2, q, q'\rangle \in Act$ **where** $InstMap[i_2] == \mathcal{T}$ **in**
16: $\quad\quad\quad Prec.\texttt{add}(n_2 \overset{r}{\twoheadrightarrow} n_1)$
17: $\quad\quad\quad$ **let** n_1' **where** $n_1 \longrightarrow n_1'$ **in**
18: $\quad\quad\quad\quad$ **repeat**
19: $\quad\quad\quad\quad\quad$ **let** $n_2' = \langle i_2, q', q''\rangle$ **where** $n_2 \longrightarrow n_2'$ **in**
20: $\quad\quad\quad\quad\quad\quad$ **if** $q' \neq \varepsilon \ \wedge \ r \in \mathcal{T}.\mathbf{P}(q')$ **then**
21: $\quad\quad\quad\quad\quad\quad\quad n_2 = n_2'$
22: $\quad\quad\quad\quad$ **until** $q'' == \varepsilon \ \vee \ r \notin \mathcal{T}.\mathbf{P}(q')$
23: $\quad\quad\quad\quad Prec.\texttt{add}(n_1' \overset{r}{\dashrightarrow} n_2)$

every dependency arc $\overset{r}{\multimap}$, built during component selection, a pair of $\overset{r}{\twoheadrightarrow}$ and $\overset{r}{\dashrightarrow}$ arcs. In particular, lines 17-23 apply a "relaxation" of the $\overset{r}{\dashrightarrow}$ arc. If port r is provided also by successor states, then we can relax the constraint imposed by the $\overset{r}{\dashrightarrow}$ arc by setting its destination to the last successor node that still provides r.

Maximally parallel plan The abstract plan is arguably the most important representation in our approach because it serves two purposes. First, it represents an intermediate, essential, step towards synthesis of the actual deployment plan. Second, it is *per se* significant, as it bears all the relevant information on the dependency order among actions of the instances involved in the deployment process.

The insight is that this representation of a plan may be seen as a *distributed concurrent plan*. Distributed, because each line in the abstract plan, specifying for each instance its life cycle, might be interpreted as a *local plan*: each instance might receive a local piece of the global plan. Concurrent, because instances may act concurrently as prescribed only by the local information. Moreover, this local plan is enriched with synchronization points that enforce an instance to stop because it must wait for a port not yet provided. Instances, thus, coordinate their actions according to the precedence constraints. One can conceive a messaging service enabling communication between instances to deal with information on reaching a given state. An instance i, for example, may have to stop waiting for someone else, j, to reach a given state s. Instance j, upon performing an $\langle \mathsf{i}, \mathsf{r}, \mathsf{s} \rangle$ action, will send a message to i informing it that it has reached s. Instance j may then proceed forward (if it does not need to wait for other messages from different instances).

As the number of synchronization points is minimal [7], the abstract plan allows for the highest degree of parallelism since each instance can act in parallel, according to its own local plan and synchronizing with the others only when strictly required. This new concept of plan shares some commonalities with the formalism of *Message Sequence Charts* [43], used to specify the behaviour of a system by visually describing the interaction among the entities that compose it.

6.2.3 Plan synthesis

The actual deployment plan that we seek is a sequence of actions on concrete instances. The main idea for the synthesis of a concrete deployment plan is to perform

[7] Minimal up-to to the choices performed in the component selection phase, which are based on heuristics.

an *adaptive topological sort* of the abstract plan until the target component is visited. The topological order ensures that actions are added to the plan in such a way that every requirement (of each component employed) activated during deployment is satisfied (i.e. configuration correctness is preserved along the way). By adaptive, we mean that the abstract plan could be rearranged during the topological sort, if component duplication is needed. Component duplication is a technique used to deal with those cases in which the abstract plan contains cycles and hence a plain topological sort is not defined. The presence of cycles means that more instances of the same component type must be deployed at the same time, in different states, in order to enact different ports.

Visiting a node is translated into several actions to be performed: a *stateChange* action and some *bind* and *unbind* actions. For each node $n_1 = \langle z_1, q_1, q'_1 \rangle$ we apply the following "translation":

unbind for each outgoing $\dashrightarrow$ arc s.t. $n_1 \overset{r}{\dashrightarrow} n_2 = \langle z_2, q_2, q'_2 \rangle$, an $unbind(r, z_1, z_2)$ action is added to the plan;

bind for each outgoing $\twoheadrightarrow$ arc s.t. $n_1 \overset{p}{\twoheadrightarrow} n_3 = \langle z_3, q_3, q'_3 \rangle$, a $bind(p, z_1, z_3)$ action is added to the plan;

stateChange action $stateChange(z_1, q_1, q'_1)$ is added to the plan.

As an example, starting from the abstract plan of Figure 7.1, a possible deployment plan for the running example is the following one:

First the three components are created. Then, master can reach state serving by means of two *stateChange* actions. The subsequent state, auth, requires interface *slave_ip*. In order to ensure this, the corresponding binding is built and slave switches to inst, providing *slave_ip*. Now master can safely move to state auth. The plan proceeds with master changing state to dump, that provides interface *dump*. The binding on port *slave_ip* is deleted and a new binding on port *dump* is built in order to enable slave to switch to state dump. Now both slave and master can reach states serving and master serving, respectively. Before this step a binding is built between

```
Plan[1] = [Create instance slave:Slave]
Plan[2] = [Create instance master:Master]
Plan[3] = [Create instance application:Application]
Plan[4] = [master : change state from uninst to inst]
Plan[5] = [master : change state from inst to serving]
Plan[6] = [slave : change state from uninst to inst]
Plan[7] = [slave : bind port slave_ip to master]
Plan[8] = [master : change state from serving to auth]
Plan[9] = [master : change state from auth to dump]
Plan[10] = [master : unbind port slave_ip from slave]
Plan[11] = [master : bind port dump to slave]
Plan[12] = [slave : change state from inst to dump]
Plan[13] = [slave : change state from dump to serving]
Plan[14] = [slave : unbind port dump from master]
Plan[15] = [slave : bind port s_mysql to application]
Plan[16] = [master : change state from dump to master serv.]
Plan[17] = [master : bind port mysql to application]
Plan[18] = [application : change state from uninst to inst]
```

each of them and application: one on *s_mysql* and the other on *m_mysql*. Finally, application can change state to inst, which is the target state.

Algorithm 5 builds the final deployment plan adding actions to the plan represented as a list, represented by variable *Plan*. Following the topological order, specified by the precedence arcs of the abstract plan, nodes become available when they do not have precedence constraints, i.e. incoming arcs. Variable $ToVisit$ represents a stack onto which nodes are pushed as soon as they have no more incoming arcs. Function `no_incoming_edges` is used to check this condition and if it's true nodes are added to the stack $ToVisit$. The algorithm relies on three auxiliary procedures, processInstanceEdge, processBlueEdges and processRedEdges, aimed at dealing respectively with $\longrightarrow$, $\twoheadrightarrow$ and $\dashrightarrow$ edges, the three kinds of edges present in the abstract plan. Given a node of the abstract plan, procedures processRedEdges and processBlueEdges deal with its outgoing $\dashrightarrow$ and $\twoheadrightarrow$ arcs. They add *unbind* and *bind* actions to the *Plan* list and remove the corresponding arcs from

Algorithm 5 Plan synthesis

```
1: Plan = [ ]; ToVisit = [ ]; finished = false;
2: for all n = ⟨i, x, y⟩ ∈ Act do                                          ▷ add initial nodes
3:   if no_incoming_edges(n) then
4:     Plan.append(new(i : InstMap[i]));
5:     ToVisit.push(n);
6: repeat
7:   repeat
8:     ⟨i, x, y⟩ = ToVisit.pop();                                          ▷ extract node
9:     if x == ε then                                                      ▷ initial node
10:      PROCESSINSTANCEEDGE(⟨i, x, y⟩)
11:    else if y == ε then                                                 ▷ final node
12:      Plan.append(del(i));
13:    else                                                                ▷ internal node
14:      Plan.append(stateChange(⟨i, x, y⟩));
15:      PROCESSREDEDGES(⟨i, x, y⟩)
16:      PROCESSBLUEEDGES(⟨i, x, y⟩)
17:      PROCESSINSTANCEEDGE(⟨i, x, y⟩)
18:    if InstMap[i] == T_target ∧ y == q_target then finished = true;     ▷ target is found
19:    Act.remove(⟨i, x, y⟩);
20:  until ToVisit == [ ] ∨ finished
21:  if ¬finished then
22:    n = Duplicate();
23:    ToVisit.push(n);
24: until finished
25: procedure PROCESSINSTANCEEDGE(⟨i, x, y⟩)
26:   let n ∈ Act where ⟨i, x, y⟩ ⟶ n ∈ Prec in
27:     Prec.remove(⟨i, x, y⟩ ⟶ n);
28:     if no_incoming_edges(n) then ToVisit.push(n);
29: procedure PROCESSBLUEEDGES(⟨i, x, y⟩)
30:   for all ⟨i, x, y⟩ -r->> ⟨i′, x′, y′⟩ ∈ Prec do
31:     Plan.append(bind(r, i, i′)); Prec.remove(⟨i, x, y⟩ -r->> ⟨i′, x′, y′⟩);
32:     if no_incoming_edges(⟨i′, x′, y′⟩) then ToVisit.push(⟨i′, x′, y′⟩);
33: procedure PROCESSREDEDGES(⟨i, x, y⟩)
34:   for all ⟨i, x, y⟩ --r--> ⟨i′, x′, y′⟩ ∈ Prec do
35:     Plan.append(unbind(r, i′, i)); Prec.remove(⟨i, x, y⟩ --r--> ⟨i′, x′, y′⟩);
36:     if no_incoming_edges(⟨i′, x′, y′⟩) then ToVisit.push(⟨i′, x′, y′⟩);
```

the abstract plan. Moreover, if the removal of an arc makes a node visitable, they add it to the $ToVisit$ stack. Similarly, procedure PROCESSINSTANCEEDGE removes the precedence arc $\longrightarrow$, adding its target node to the $ToVisit$ stack if it has no incoming arcs.

At the beginning, all initial nodes are pushed on $ToVisit$ (lines 2-5) and a *new* action is added to the plan for every initial node (line 4). The algorithm then proceeds considering one action $a = \langle i, x, y \rangle$ in $ToVisit$ at a time, until the target node is encountered or $ToVisit$ becomes empty.

If a is an initial node, its outgoing precedence arcs are removed by calling procedure processInstanceEdge (line 9). In case a is, instead, a final node, a corresponding *del* action is added to the plan (line 12).

Finally, if a is an intermediate node, a *stateChange* action is added to the plan (line 14). The a outgoing red, blue and precedence arcs are then removed from the abstract plan by calling in sequence the auxiliary procedures processRedEdges, processBlueEdges, and processInstanceEdge (lines 15-17). At the end of the inner loop, variable $finished$ is set to true if the target node is encountered (line 18) and node a is removed from the abstract plan (line 19).

The above translation exploits the fact that in the model *bind* and *unbind* actions are allowed, disregarding the active state: a binding may be built even between inactive ports. [8] In fact, Algorithm 5 specifies that the *bind* action is performed before the requirer instance reaches the state that activates the requirement.

The final part, lines 21-23, by calling function Duplicate in Algorithm 6, deals with the duplication process, explained below.

Duplication

This technique is employed when the abstract plan is a cyclic graph. Let us consider, for instance, a slight modification of the running example in which the application's architecture demands a secondary component of type Master in state serving. This new scenario may be modeled by modifying the Application component in a way that the target installed state has an additional required port, *mysql* (provided by state

[8] Configuration correctness is not hard-coded in the actions but it is required at the level of deployment plan. In such a plan all intermediate configurations must be correct.

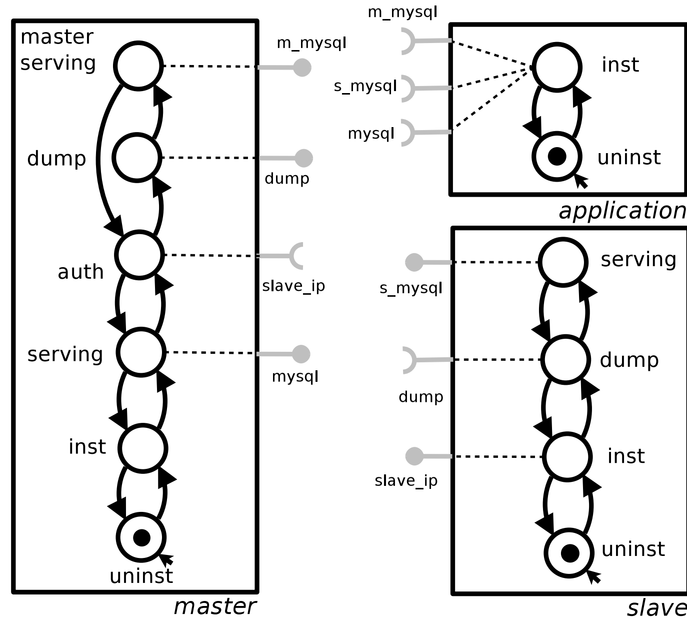


Figure 6.10: Initial configuration for the new scenario.

serving of component type Master). The Aeolus$^-$ model for the new architecture description is depicted by Figure 6.10.

The resulting abstract plan is shown in Figure 6.11 where nodes forming a cycle are highlighted in red and tagged by identifiers. In this new plan there are two nested cycles, one containing the other: one formed by nodes $\{1, 2, 3, 4, 5, 6, 7\}$ and a smaller one formed by nodes $\{1, 2, 3, 4, 7\}$. A new pair of arcs, highlighted in bold, is added as an effect of the new required *mysql* port: $\xrightarrow{mysql}$ and $\overset{mysql}{\dashrightarrow}$. Notice that the arc "responsible" for introducing the cycle is $\langle a, I, \varepsilon\rangle \overset{mysql}{\dashrightarrow} \langle m, S, A\rangle$. We will see that it is precisely the arrival node $\langle m, S, A\rangle$ the one to be used in the duplication procedure.

In this case the topological visit is unable to reach the target as the abstract plan is a cyclic graph. This happens when an instance z is required to be in two states q and q' at the same time as they enact different functionalities (ports) simultaneously demanded. It is then necessary to duplicate that instance: a new instance z' that will stay in state q is created, thus keeping the provided port active, and in this way the original instance is allowed to perform the $stateChange(z, q, q')$ action. For

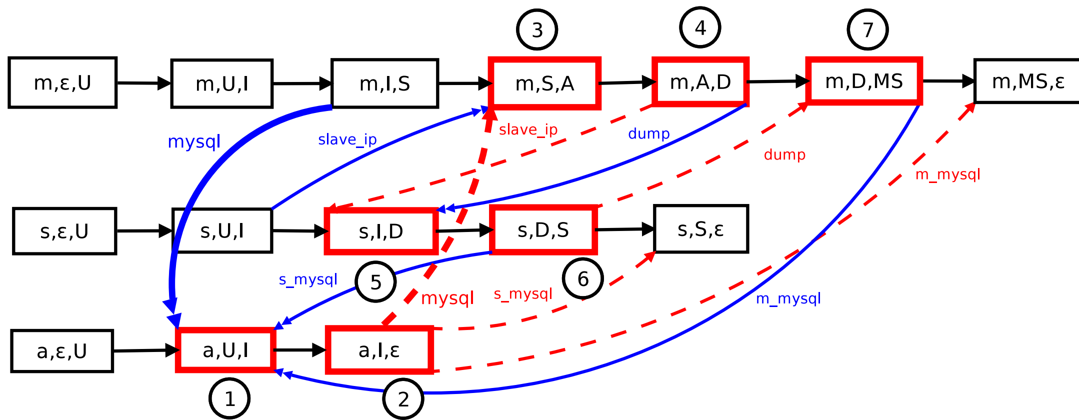


Figure 6.11: Cyclic abstract plan for the modified running example.

this reason the technique used to deal with this issue takes the name of *duplication*. A cycle is "solved" by creating an additional instance line and moving some arcs. Figure 6.12 depicts the effect of applying the duplication procedure on the previous cyclic abstract plan: a duplicate instance m', of type Master, is created and its life cycle stops in state serving, keeping providing port *mysql*. The $\overset{mysql}{\twoheadrightarrow}$ and $\overset{mysql}{\dashrightarrow}$ arcs have been moved towards the duplicate instance.

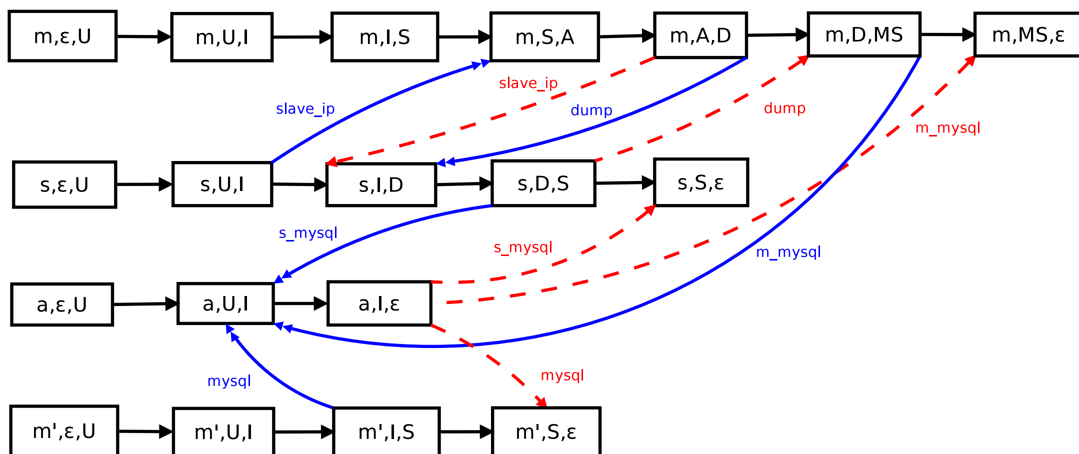


Figure 6.12: Abstract plan after duplication.

To find the instance to be duplicated we rely on finding, among the nodes forming the cycle, the one with only incoming $\dashrightarrow$ arcs. The soundness of this principle is guaranteed by a formal result, Lemma 6.2, stating that every cycle contains at least

such a node. An $r \overset{k}{\dashrightarrow} p$ arc points towards the provider p of port k. By duplicating the instance corresponding to node p we make sure that the pair of $\overset{k}{\twoheadrightarrow}$, $\overset{k}{\dashrightarrow}$ arcs is redirected towards the duplicate instance and the cycle is broken.

The advantage of this approach is twofold. First of all, it enables to deal uniformly with complex cycles: nesting of cycles is allowed (as is the case in our example). Second, there is no need to perform any cycle detection as a pre-processing phase, which would be computationally costly. The abstract plan's rearrangement, namely the introduction of new duplicate instances, can be carried out on-the-fly. The abstract plan is topologically sorted and if at the end stack $ToVisit$ is empty but target was not visited (see lines 20-21 in Algorithm 5), then we know that there must be a cycle and duplication is needed.

Algorithm 6 Duplication

1: **function** DUPLICATE
2:   **let** $n = \langle i, x, y \rangle \in Act$ **where** $y \neq \varepsilon \wedge \nexists n' \in Act \,.\, (n' \longrightarrow n \in Prec \vee n' \overset{r}{\twoheadrightarrow} n \in Prec)$ **in**
3:     i' = `getFreshName`(); $InstMap[i'] = InstMap[i]$;
4:     Act.`add`($\langle i', x, \varepsilon \rangle$);
5:     **for all** $n' \overset{r}{\dashrightarrow} \langle i, x, y \rangle \in Prec$ **do**
6:       $Prec$.`remove`($n' \overset{r}{\dashrightarrow} \langle i, x, y \rangle$); $Prec$.`add`($n' \overset{r}{\dashrightarrow} \langle i', x, \varepsilon \rangle$);
7:     **for** $(j = Plan.size() - 1; j \geq 0; j = j - 1)$ **do**
8:       **if** $Plan[j] == bind(r, i, z)$ **then** $Plan[j] = bind(r, i', z)$;
9:       **else if** $Plan[j] == bind(r, z, i)$ **then** $Plan$.`insert`($bind(r, z, i'), j$);
10:       **else if** $Plan[j] == unbind(r, i, z)$ **then** $Plan[j] = unbind(r, i', z)$;
11:       **else if** $Plan[j] == unbind(r, z, i)$ **then**
12:         $Plan$.`insert`($unbind(r, z, i'), j$);
13:       **else if** $Plan[j] == new(i : \mathcal{T})$ **then** $Plan$.`insert`($new(i : \mathcal{T}), j$);
14:       **else if** $Plan[j] == stateChange(\langle i, x, y \rangle)$ **then**
15:         $Plan$.`insert`($stateChange(\langle i', x, y \rangle, j$);
16:     **return** $\langle i, x, y \rangle$;

The Duplicate procedure is detailed by Algorithm 6. It starts by identifying a state change node $\langle i, x, y\rangle$ with only incoming $\dashrightarrow$ arcs (line 2). i is the instance to duplicate until the node preceding $\langle i, x, y\rangle$. A fresh name i' is forged to identify the new instance (line 3). A final node $\langle i', x, \varepsilon\rangle$ for i' is added to the set of actions (line 4). Node $\langle i', x, \varepsilon\rangle$ must be final because the new instance is intended to stop into state x. All $\dashrightarrow$ arcs incoming into $\langle i, x, y\rangle$ are redirected towards the new $\langle i', x, \varepsilon\rangle$ node (lines 5-6). Then, the actions already performed on i are duplicated in order to perform them also on the new instance i' (lines 7-15). Actions *new* and *stateChange* of i' are added to the plan immediately after the *new* and *stateChange* actions of i (lines 13, 15). Actions *bind* and *unbind* where i requires an interface provided by another instance, are replicated (lines 9, 12). The *bind* and *unbind* actions where i, instead, is acting as a provider for other instances, are replaced with *bind* and *unbind* actions involving i' instead of i (lines 8, 10). Finally, node $\langle i, x, y\rangle$ is the return value; notice that this node may be immediately added to the $ToVisit$ stack (line 23) since all its incoming precedence constraints have been removed by the duplication procedure.

Termination of Algorithm 5 is guaranteed by the fact that the number of duplications needed to reach the target component is bound by the number of nodes in the original abstract plan.

6.2.4 Heuristics

In order to efficiently compute a deployment plan, our approach relies on the use of heuristics in the component selection part (see Algorithm 3) during the abstract planning phase.

The aim of the proposed technique is to find a deployment plan. The intended goal, however, is to try to synthesize a plan that is minimal w.r.t. the number of components deployed. Heuristics are employed in order to overcome the computational complexity incurred in, when trying to find the *optimal* solution, that is a deployment plan which uses exactly the minimum number of components.

Component selection involves two kinds of choice: one for the origin node and one for the node provider of a given required port. The metrics employed by the heuristics in use, rely on three parameters:

cardinality the total number of required ports enabled along the path from the root to a certain node in the reachability graph construction;

distance the number of *stateChange* actions needed to reach a node from the root of the reachability graph;

fan-in the number of incoming ⇢ arcs to the node considered from nodes that have already been selected.

The first and second parameter estimate the "cost" of reaching a given node: the former in terms of how many requirements must be fulfilled along the way to reach it, the latter in terms of number of actions. The third parameter estimates, instead, the fitness of a node selection in terms of the number of nodes for which the given node can potentially act as a provider. The insight is that preference should be given to the node that can satisfy the requirements of the highest number of nodes, in the hope that, as an effect, less nodes (hence less components) will be needed.

From the algorithmic point of view, the first two values may be computed (top-down) during reachability graph construction, while the second can be computed (bottom-up) while performing component selection.

Table 6.2 summarizes the heuristics defined as a starting point. The order in the table reflects the order of precedence using a lexicographic ordering. For the origin's choice, for instance, the node with the highest fan-in value is chosen, in case of tie we search for the node with minimum cardinality value, in case of ties we choose a node that is already present and last we choose based on the minimum distance value. The procedure for the choice of the provider follows the same principle.

The evaluation of the impact of variations on the precedence order of parameters employed, by alternative heuristics, deserves further investigation and is left to future work, as explained in Chapter 8.

Origin's choice	**Provider's choice**
1. max fan-in 2. min cardinality 3. copy 4. min distance	1. max fan-in 2. min cardinality 3. min distance

Table 6.2: Parameters' precedence order for the heuristics employed.

6.3 Formal analysis

This section is dedicated to the formal analysis of the proposed technique. First, it is proven to be sound and complete. This result guarantees that algorithm *DeploymentPlanner* always answers and when it does the answer is correct. Afterwards, the algorithm's efficiency is supported by proving that it has polynomial computational complexity.

6.3.1 Soundness & completeness

In this section we prove that algorithm *DeploymentPlanner* produces a deployment plan if and only if the target state of the given component is reachable. The proof is split in the two directions of implication in the following two theorems. We start by presenting the soundness part: the algorithm always generates deployment plans that are correct.

Theorem 6.1 (Soundness)**.** *Given a universe of components U, a component type $\mathcal{T}_t$, and a target state q_t, if the DeploymentPlanner algorithm computes a sequence of actions $\mathsf{A} = \alpha_1, \ldots, \alpha_m$, then $\langle U, \emptyset, \emptyset, \emptyset \rangle \xrightarrow{\alpha_1} \mathcal{C}_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_m} \mathcal{C}_m$ is a deployment plan for $\mathcal{T}_t$ in state q_t.*

Proof. Suppose that the output of *DeploymentPlanner* is $\mathcal{C}_0 \xrightarrow{\alpha_1} \mathcal{C}_1 \xrightarrow{\alpha_2} \dots \xrightarrow{\alpha_m} \mathcal{C}_m$ where $\mathcal{C}_0 = \langle U, \emptyset, \emptyset, \emptyset \rangle$. In order to prove the thesis we have show that the following conditions hold:

1. $\mathcal{C}_0$ is correct;
2. $\mathcal{C}_m$ contains an instance z with $S(z) = \langle \mathcal{T}_t, q_t \rangle$;
3. each reconfiguration preserves configuration correctness.

Condition 1. is satisfied since the initial configuration $\mathcal{C}_0$ is trivially correct as it contains only instances in their initial state (hence no required port is activated).

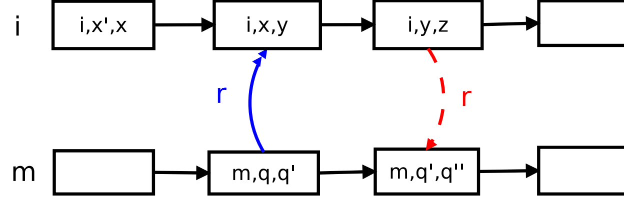
As for condition 2., if *DeploymentPlanner* outputs a plan, it means it has terminated without raising an exception. Thus, target $\langle \mathcal{T}_t, q_t \rangle$ is generated while building the reachability graph and the final configuration $\mathcal{C}_m$ contains an instance z of component type $\mathcal{T}_t$ in state q_t. This holds because Algorithm 5 terminates when node $\langle z, q, q_t \rangle$ is visited. As an effect $stateChange(z, q, q_t)$ action is added to the plan and the final state of component z is thus q_t.

It remains to show that condition 3. above holds. This amounts to prove that for every $1 \leq j \leq m$ if $\mathcal{C}_{j-1}$ is correct and $\mathcal{C}_{j-1} \xrightarrow{\alpha_j} \mathcal{C}_j$, then $\mathcal{C}_j$ is correct. We work by cases on the kind of α_j action. If α_j is a *bind* or *new* action, correctness is trivially preserved since these actions do not violate any requirement.

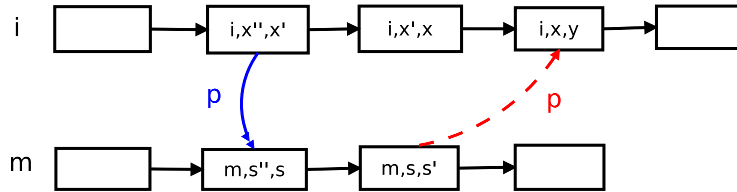
If $\alpha_j = stateChange(i, x, y)$, correctness may be invalidated in two ways: either state y of i requires a port r, not provided in $\mathcal{C}_j$ or i in state y stops providing a port p, needed by someone else. We will show that neither of these situations can occur.

In the first case, if m is the name of the instance providing r, the situation can be depicted as follows:

Action $stateChange(i, x, y)$ must have been added to the plan by visiting a node $\langle i, x, y \rangle$ in the abstract plan (line 14 in Algorithm 5). In the former case, it suffices to show that there is an incoming arc into $\langle i, x, y \rangle$ from a node $n_r = \langle m, q, q' \rangle$ where q' provides interface r, required by i in state y. In fact, its existence would



guarantee that node n_r is visited before, adding actions $stateChange(m, q, q')$ and $bind(r, m, i)$ to the plan, and so the requirement on r would be already fulfilled when i enters state y. Let's show that such an arc actually exists. In the reachability graph construction, node $\langle \mathcal{T}, y \rangle$, corresponding to $stateChange(i, x, y)$, is added only after making sure its requirements are fulfilled (line 7 in Algorithm 2). Consequently an $\longrightarrow$ arc is added (line 14 in Algorithm 2) for each node $\langle \mathcal{T}, w \rangle$ predecessor of $\langle \mathcal{T}, y \rangle$ (i.e. $(w, y) \in \mathcal{T}.\texttt{trans}$). Among $\langle \mathcal{T}, y \rangle$'s predecessors there must be also $\langle \mathcal{T}, x \rangle$. During component selection phase, for every $\longrightarrow$ arc, a provider node is chosen for all requirements (line 9 in Algorithm 3) and hence also for r. Let's call this provider z. An $z \overset{r}{\multimap} \langle \mathcal{T}, y \rangle$ arc is thus added to keep track of this choice (line 11 in Algorithm 3). Finally, in Algorithm 4, for every arc in $SReq$, among which also $z \overset{r}{\multimap} \langle \mathcal{T}, y \rangle$, an $\overset{r}{\twoheadrightarrow}$ arc going from $n_r = \langle m, q, q' \rangle$ to $\langle i, x, y \rangle$ is added (lines 13-16). We have shown that there is a an incoming $\overset{r}{\twoheadrightarrow}$ arc into node $\langle i, x, y \rangle$.
Configuration's correctness could also be violated by $stateChange(i, x, y)$ ceasing to provide a needed interface r. In this case, if m is the name of the component requiring port p, the situation is the following:



Similarly as above, if there is an incoming $\overset{p}{\dashrightarrow}$ arc from a node $n_p = \langle m, s, s' \rangle$ where s is the state requiring p, provided by i in state x, then correctness is not violated as the state transition from s to s' will take place before i leaves state x. By hypothesis we know that instance i has been chosen as a provider of interface

p for some other component m. If $\mathcal{T}''$ is the component type of m, some state s of his requires interface p and by reasoning in the same way as above we are sure that a $\overset{p}{\twoheadrightarrow}$ arc is added from a vertex of instance i to one of instance m. Now, because of the way pairs of $\twoheadrightarrow$ and $\dashrightarrow$ arcs are added, the destination node of arc $\overset{p}{\twoheadrightarrow}$ must necessarily be the predecessor of $\langle m, s, s' \rangle$, let it be $\langle m, s'', s \rangle$ for some state s''. As for the source vertex it must be the first vertex in the transitive closure of predecessors of $\langle i, x, y \rangle$, that keep providing interface p. We cannot just say it's the predecessor of $\langle i, x, y \rangle$ because relaxations are applied (lines 17-23 in Algorithm 4) . Let us denote it by $\langle i, x'', x' \rangle$. If an arc $\langle i, x'', x' \rangle \overset{p}{\twoheadrightarrow} \langle m, s'', s \rangle$ was added, then a twin one $\langle m, s'', s \rangle \overset{p}{\dashrightarrow} \langle i, x, y \rangle$ must also have been added (line 23 in Algorithm 4). Vertex $\langle i, x, y \rangle$ is the successor of $\langle i, x'', x' \rangle$ if relaxation did not affect this arc, otherwise $\langle i, x, y \rangle$ is in the transitive closure of $\longrightarrow$ arcs of vertex $\langle i, x'', x' \rangle$, where x is the last state providing p. By proving the existence of an incoming $\overset{p}{\dashrightarrow}$ into $\langle i, x, y \rangle$ we have shown that correctness is not violated by a *stateChange* action occurring in the produced plan.

Let us consider the case $\alpha_j = \mathit{unbind}(r, \mathit{prov}, \mathit{req})$, where instance *prov* provides r to instance *req*. Correctness violation occurs if in $\mathcal{C}_m$ *req* is in the same state x, requiring r, and the requirement ceases to be fulfilled. Action $\mathit{unbind}(r, \mathit{prov}, \mathit{req})$ must have been added during plan synthesis because there was an $\langle \mathit{req}, x, y \rangle \overset{r}{\dashrightarrow} \langle \mathit{prov}, v, w \rangle$ arc in the abstract plan (lines 34-36 in Algorithm 5). In case of an intermediate node, like $\langle \mathit{req}, x, y \rangle$, plan synthesis, before processing the outgoing red edges (by invoking PROCESSREDEDGES), adds action $\mathit{stateChange}(\mathit{req}, x, y)$ to the plan, guaranteeing that instance *req* has already left state x before performing action $\mathit{unbind}(r, \mathit{prov}, \mathit{req})$ (lines 14-15 in Algorithm 5). This ensures that instance *req*, that required r, has already stopped requiring it.

Similarly, if $\alpha_j = \mathit{del}(i)$, it may violate correctness by deleting a component that provides a port that is still needed. This, however, is never the case because final nodes of provider instances have an incoming $\overset{r}{\dashrightarrow}$ arc for every provided port r. This is easily seen by repeating the argument of case $\alpha_j = \mathit{stateChange}(i, x, y)$ above. Therefore, by Algorithm 5, this action is performed only after all instances

requiring r have ceased requiring it (this may happen by means of a *stateChange* or a *del* action). This concludes the proof of this case and of the whole theorem.

$\square$

The second result shows that the algorithm is complete, i.e. if a deployment plan exists, then the algorithm will eventually find one. To prove completeness we rely on few lemmas. The first one shows that all circularities in the abstract plan contain at least an $\dashrightarrow$ arc. The second lemma, based on the previous one, states, in presence of circularities, the existence of a node whose incoming arcs are all of the $\dashrightarrow$ kind. This is the node chosen by Algorithm 6 for duplication, to break the cycle and proceed with the topological visit of the abstract plan. Finally, Lemma 6.3 proves termination of Algorithm 5.

Lemma 6.1. *Every cycle in the abstract plan contains at least an* $\dashrightarrow$ *arc.*

Proof. (By contradiction). Assume that there is a cycle which contains only $\twoheadrightarrow$ and $\longrightarrow$ arcs. The situation can be exemplified in a basic case with only three arcs as in Figure 6.13.

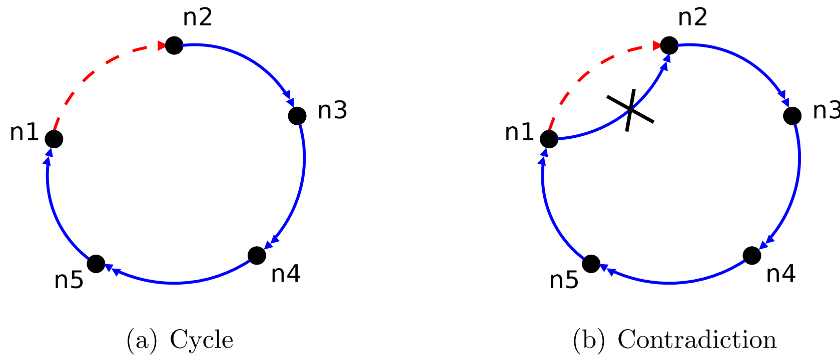
Figure 6.13: Basic scenario (only 3 arcs).

According to the reachability graph construction an $\langle i, x, y\rangle \longrightarrow \langle i, y, z\rangle$ means pair $\langle \mathcal{I}, y\rangle$ is at a lower generation level (higher up in the pyramid) than $\langle \mathcal{I}, z\rangle$, where $\mathcal{I}$ is the component type of instance i. We can denote this fact by writing $L(\langle \mathcal{I}, y\rangle) < L(\langle \mathcal{I}, z\rangle)$, where $L(p)$ is the level of pair p.
Arcs of the $\twoheadrightarrow$ kind also bear informations on the generation level. $n_1 \overset{p}{\twoheadrightarrow} n_2$ means

that pair n_1, the provider, lies at a lower generation level w.r.t. the requirer n_2 (a node always satisfies its requirements by means of nodes from one generation before). So, we have that $n_1 \overset{p}{\twoheadrightarrow} n_2$ implies $L(n_1) < L(n_2)$. Looking at the diagram above we can see that $L(\langle \mathcal{M}, v \rangle) < L(\langle \mathcal{I}, y \rangle)$ (because $\langle m, u, v \rangle \overset{p}{\twoheadrightarrow} \langle i, x, y \rangle$) and $L(\langle \mathcal{I}, z \rangle) < L(\langle \mathcal{M}, v \rangle)$ (because $\langle i, y, z \rangle \overset{p}{\twoheadrightarrow} \langle m, u, v \rangle$). This leads to a contradiction as we end up concluding that $L(\langle \mathcal{M}, v \rangle) < L(\langle \mathcal{I}, z \rangle)$ and $L(\langle \mathcal{I}, z \rangle) < L(\langle \mathcal{M}, v \rangle)$. The argument can be easily seen to hold in the general case. Hence the cycle must contain only $\twoheadrightarrow$ arcs. If this is the case the type-state pairs corresponding to the involved nodes are mutually dependent, i.e. component z_1 to reach a state q_1 needs something provided by z_2 in state q_2 and, vice versa, the component z_2 to reach q_2 needs something provided by z_1 in q_1. This is a contradiction because by Algorithm 2 mutually dependent type-state pairs cannot be produced. This concludes the proof. □

Lemma 6.2. *For each cycle in the abstract plan there exists a node whose incoming arcs are all of the* $\dashrightarrow$ *kind.*

Proof. (By contradiction). Without loss of generality, assume that there is a cycle such as the one depicted in Figure (a) below:



(a) Cycle (b) Contradiction

Suppose that there is an incoming arc into node $n2$ that is not a $\dashrightarrow$ arc. If it is a $\twoheadrightarrow$ then we end up in a situation such as the one depicted in Figure (b) above. No matter how we place the $\twoheadrightarrow$ arc, there is a cycle made of only $\twoheadrightarrow$ arcs which contradicts previous lemma. The same argument holds if we introduce a $\longrightarrow$ arc. □

Lemma 6.3. *Algorithm 5 for adaptive topological-sort terminates.*

Proof. Algorithm 5 visits all the nodes of the abstract plan. The total number of nodes in the abstract plan is finite. If there are no cycles the result is trivial. If circularities exists in the abstract plan, duplication is needed. This procedure introduces every time a new node to the abstract plan and redirects some arcs. We have to show that the number of duplications applied is bound. When duplication is invoked, a node n with only incoming $\dashrightarrow$ arcs is found, a duplicate node n' is built and all the arcs incoming into n are redirected towards n'. Hence n cannot be chosen for duplication a second time since it has no more incoming arcs. This mens that every node is duplicated at most once. Moreover, duplication removes all the cycles involving node n, without creating new ones as the duplicate node has only incoming edges and no outgoing ones. The number of duplications applied is thus bound by the total number of nodes in the starting abstract plan. □

Theorem 6.2 (Completeness)**.** *Given an universe of components U, a component type $\mathcal{T}_t$, and a target state q_t, if a solution exists to the deployment problem on input $I = (U, \mathcal{T}_t, q_t)$, then algorithm DeploymentPlanner returns a deployment plan for I.*

Proof. Since by hypothesis there is a sequence actions that allows the deployment of component $\mathcal{T}_t$ in state q_t, during reachability analysis the type-state pair $\langle \mathcal{T}_t, q_t \rangle$ is obtained. A correct plan is produced (Theorem 6.1) if the abstract plan generation (Algorithm 4) and plan synthesis (Algorithm 5) phases terminate. The former terminates because, given the reachability graph, the total number of maximal paths is finite. Termination of the latter, instead, is given by Lemma 6.3. □

6.3.2 Computational complexity

If the synthesized deployment plan should limit to the minimum the number of the employed components we end up dealing with a problem whose decisional version is in NP. Algorithm *DeploymentPlanner*, which forms the basis of a prototype planner, tackles the complexity of the problem by relying on heuristics, as detailed in

Section 6.2.4. In order to assess its effectiveness we first prove its efficiency from the computational complexity point of view. The following formal result shows that the complexity of algorithm *DeploymentPlanner* is polynomial.

Theorem 6.3 (Complexity). *The DeploymentPlanner algorithm runs in polynomial time.*

Proof. Let us denote with

- k the total number of possible component type-state pairs;
- b the maximal number of predecessors of a type-state pair;
- h the maximal number of ports required by any component's state.

Let us reason on the complexity of each step performed in the *DeploymentPlanner* algorithm.

[Reachability analysis]. Every level of the reachability graph has no more than k type-state pairs. In the worst case, each iteration in the construction of the reachability graph adds only one type-state pair. Hence the algorithm terminates and in the pyramid there are at most $k + 1$ levels. To build a new level from given one it is also necessary to filter the successors of the components in the previous level by checking if their requirements are satisfied. Since a component has at most k successors and requires at most h ports, the cost of building a level is $O(hk^2)$. The pyramid has at most $k + 1$ levels, hence Algorithm 2 runs in $O(hk^3)$ time.

[Component selection]. To select the components to use and the bindings among them, Algorithm 3 considers for every type-state pair at most h ports and b parent. Each port could be provided by any of the nodes at the previous level and hence at most hk choices must be taken into account. For each node the cost is bound of $O(b + hk)$ operations. Since each of the $k + 1$ levels contains at most k pairs, the total number of pairs in the reachability graph is $O(k^2)$. As a result Algorithm 3 takes $O((b + hk)k^2)$ time.

[Abstract plan generation]. After component selection in the reachability graph there are at most k^2 maximal paths of length k. This holds because for every

node $\langle \mathcal{T}, q \rangle$ there is exactly a single origin $\langle \mathcal{T}, q' \rangle$, corresponding to the choice of state q' among the predecessors of state q in the component type $\mathcal{T}$. [9] Hence the computation of the maximal paths in Algorithm 4 is bound by $O(k^3)$. The generation of the instance lines costs at most $O(hk^3)$ since the abstract plan contains $O(k^3)$ nodes, each of them having no more than $h+1$ outgoing precedence arcs (h $\dashrightarrow$ arcs + 1 $\longrightarrow$ arc). Algorithm 4 has thus complexity $O(k^3) + O(hk^3) = O(hk^3)$.

[Plan synthesis]. If the abstract plan does not contain cycles the adaptive topological sort does not involve duplication. In this case the cost is simply bound by the total number of nodes in the abstract plan, $O(k^3)$ (topological sort is linear w.r.t. the number of nodes). If cycles appear in the abstract plan, Algorithm 5 duplicates a node whenever the topological visit gets stuck. In the worst case, a duplication is needed for every node of every instance and to find which node to duplicate one may need to visit all the nodes. Every node n has at most hk^2+h+1 incoming arcs: hk^2 $\dashrightarrow$ arcs coming from the fact that pair $\langle \mathcal{T}, q \rangle$ in the abstract plan, corresponding to n, may work as a provider for at most k^2 other pairs where each of them has h requires to be fulfilled; h $\twoheadrightarrow$ arcs come from the fact that n itself may require at most h ports; 1 is the instance arc $\longrightarrow$. Finding the node to be duplicated has thus a worst case cost of $O(k^3) \cdot O(hk^2 + h + 1) \cong O(k^3) \cdot O(hk^2) = O(hk^5)$. The duplication procedure updates the abstract plan by adding a node for every node in the starting plan and it redirects all the arcs incoming into the original node towards the duplicate. The cost of redirection for a single duplication is thus $O(hk^2)$ (redirect at most hk^2 $\dashrightarrow$ arcs). The total number of duplications is bound by k^2, the number of nodes in the reachability graph. To see why this holds, consider that if we duplicate each pair $\langle \mathcal{T}, q \rangle$ we would end up in an abstract plan where each instance has a duplicate for each of the states that it traverses. Each of these duplicate instances stops in a given state, working as a provider for all other pairs that require ports provided by it. Therefore, in the worst case, the cost of all duplications is $O(k^2) \cdot O(hk^5) = O(hk^7)$.

[9] This means that, after component selection, each node in the reachability graph may be reached by exactly one and only one path.

[Total cost]. Summing up all the contributions, the *DeploymentPlanner* algorithm has a total complexity of $O(hk^3) + O((b + hk)k^2) + O(hk^3) + O(hk^7)$, which considering b bound by k, amounts to $O(hk^7)$. □

Chapter 7

Practice

This chapter deals with the development of a prototype planner putting into practice the ideas of the technique presented in previous chapter.

7.1 METIS: a deployment planner

In order to assess the effective viability of the proposed approach, a proof of concept implementation of a tool for synthesizing deployment plans has been developed. METIS, which stands for Modern Engineered Tool for Installing Software systems, is an ad-hoc planner that implements the algorithms presented/defined in Section 6.2. Namely, starting from a pool of available component types and a configuration, it generates a sequence of actions necessary to deploy such a system.

METIS is invoked on the command-line by issuing the following command:

```
./metis.native -u universe.json -c Application -s Inst
  -o plan.txt -ap abstract-plan.dot
```

The input passed to METIS consists of:

- a universe, corresponding to a file with the available component types (option `-u`);
- a component type, which identifies the target (option `-c`);
- a target state of the previous component type (option `-s`);

- a text file, used to store the sequential plan output (option `-o`);
- a `dot` file, used to store the abstract plan output (option `-ap`).

Input

The universe is simply a list of component types. Each component type is specified in JSON format [5]. As an example consider Listing 7.1 showing an excerpt from the universe input for the running example. In the code snippet we can identify two component types: Application and Slave. Each component type is specified by means of a name (field `cname`) and by the automaton describing its behaviour. Each state is defined by a `name`, a list of `successors`, a list of the provided ports (`provides`) and a list of the required ones (`requires`).

Output

The output consists in two parts:

1. the *sequential plan* as a sequence of actions, needed to deploy the target. This plan is in textual format;
2. the *abstract plan* as a directed graph in `dot` format [52, 42].

Listing 7.2 depicts the output for the running example in the case where no duplication is needed. Each action is prefixed with the corresponding instance name. Figure 7.1 shows the abstract plan obtained for the running example.

Source code

METIS is developed as open source code freely available from GitHub at the following address `https://github.com/aeolus-project/metis`. The implementation is about 3.5K lines of code written in OCaml and it is distributed under GPL license.

Listing 7.1: Sample input

```
[
  {
    "cname" : "Application",
    "automaton" :
      [
        {
          "name" : "Uninst",
          "successors" : ["Inst"],
          "provides" : [],
          "requires" : []
        },
        {
          "name" : "Inst",
          "successors" : ["Uninst"],
          "provides" : [],
          "requires" : ["m_mysql", "s_mysql"]
        }
      ]
  },
  {
    "cname" : "Slave",
    "automaton" :
      [
        {
          "name" : "Uninst",
  ...
]
```

7.2 Validation

In Chapter 4 we introduced a general definition for the class of planning problems and we argued that the deployment problem is naturally modeled as an instance of this class.

Listing 7.2: Sequential plan output for the running example.

```
Plan[1] = [Create instance slave:Slave]
Plan[2] = [Create instance master:Master]
Plan[3] = [Create instance application:Application]
Plan[4] = [master : change state from uninst to inst]
Plan[5] = [master : change state from inst to serving]
Plan[6] = [slave : change state from uninst to inst]
Plan[7] = [slave : bind port slave_ip to master]
Plan[8] = [master : change state from serving to auth]
Plan[9] = [master : change state from auth to dump]
Plan[10] = [master : unbind port slave_ip from slave]
Plan[11] = [master : bind port dump to slave]
Plan[12] = [slave : change state from inst to dump]
Plan[13] = [slave : change state from dump to serving]
Plan[14] = [slave : unbind port dump from master]
Plan[15] = [slave : bind port s_mysql to application]
Plan[16] = [master : change state from dump to master serv.]
Plan[17] = [master : bind port mysql to application]
Plan[18] = [application : change state from uninst to inst]
```

In absence of benchmarks specific for the deployment problem the validation of the METIS tool has been conceived by means of synthetic test cases taking into account different aspects of the problem. An encoding of the deployment problem into the Planning Domain Definition Language (PDDL) was defined in order to verify if it can be dealt with standard planning techniques. PDDL is the standard language for describing planning problems.

Several experiments were run both with standard planners and METIS on a test suite. The results obtained are encouraging as our planner is able to deal with instances of the problem with hundreds of components in less than a minute while the feasible instance size with standard planners is in the order of dozens.

In the following sections we first describe the encoding and then we report on the experimental results.

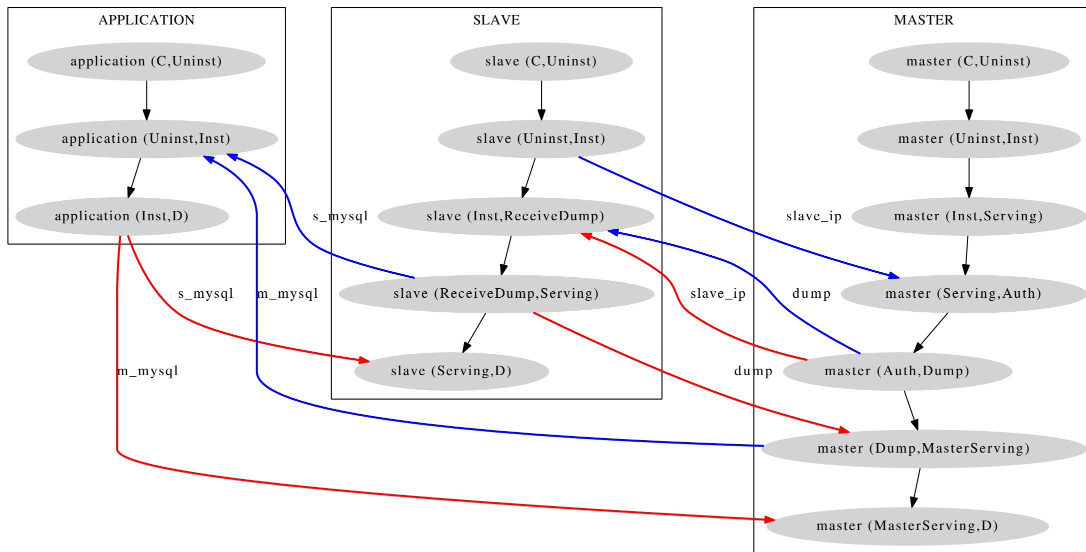


Figure 7.1: Abstract plan output for the running example.

Deployment problem as a planning problem

In the context of knowledge representation and reasoning the proposals on representing and reasoning about actions and change and, more specifically, for the problem of planning [18], have relied on the use of concise and high-level languages, commonly referred to as *action description languages* [37]. Action languages allow one to write propositions that describe the effects of actions on states, and to create queries to infer properties of the underlying transition system. In 1998, a declarative language for planning has been defined, establishing a common syntax in order to allow different research groups to test their solvers. This language is known as PDDL and his last release is 3.1 [35] (see [58, 38] for information on planning competitions and PDDL). To do so we have defined an encoding of our specific planning problem into PDDL: each component instance is translated into one PDDL object with possible actions corresponding to state changes. Such actions are enabled only when the other objects in the configuration provide the required interfaces. The encod-

ing abstracts from the *bind* and *unbind* actions [1] and limits the number of objects that may be concurrently used. This limitation is necessary because all the solvers assume a finite number of objects: without this limitation the planning problem is undecidable. In the experiments this parameter was set to the minimum value as the computation time increases exponentially w.r.t. it.
In PDDL, the specification of a planning problem is split in two parts:

1. the *domain file* for predicates and actions;

2. the *problem file* for objects, initial state and goal specification.

We will sketch the encoding of the deployment problem by presenting (part of) the two files. The full encoding of the problem into the PDDL language is available at [9].

Domain part

The *domain* is specified in terms of objects, predicates, operators/actions. W.r.t. to the formal definition of planning problem given in Chapter 4:

- objects are constant symbols and represent the things that one is interested in. In our case we have: components, ports (and nodes, explained below);

- predicates are properties of objects. For example a predicate may state whether a component provides a given port or not;

- operators/actions are *stateChange*, *new*, *del*.

An object called *node* must be introduced. Intuitively it can be considered as a *place* where one and only one component could be in a given state. Any component in use must be deployed in one of this nodes.

[1] Both *bind* and *unbind* actions can be added in a post processing phase to form a valid deployment plan in polynomial time.

Predicates The predicates employed in the encoding are given in Listing 7.3.

Listing 7.3: Predicates

```
1  (:predicates
2
3    (initial_component ?c - component)
4      ; intial state of the component c
5
6    (transition ?c1 - component ?c2 - component)
7
8    (component_provides_port ?c - component ?p - port)
9    (component_requires_port ?c - component ?p - port)
10
11   (node_component ?n - node ?c - component)
12
13   (used_node ?n - node)
14
15 )
```

A component c represents here a type-state pair $\langle \mathcal{T}, q \rangle$.
(initial_component c) says that component c is in its initial state. *(transition c1 c2)* says that there is a transition from $c1$ to $c2$.
(component_provides_port c p) and *(component_requires_port c p)* hold respectively when c provides or requires port p. *(used_node n)* is employed to check if node n is free or not.

Actions Each action is naturally specified by a list of parameters, the preconditions and the effects.

The *new* action can be viewed as assigning a component in a given state to an empty node. The PDDL code to describe this can be seen in Listing 7.4.

Listing 7.4: *new* action

```
1  (:action create_node
2    :parameters (?n - node ?c - component )
3    :precondition (and
4      (not (used_node ?n))
5      (initial_component ?c)
6    )
7    :effect (and
8      (node_component ?n ?c)
```

```
9        (used_node ?n)
10     )
11   )
```

This action requires two parameters: a component c and a node n. It can fire if n is not used (i.e. the predicate *used_node* is false for n) and if c is in its initial state. When the *new* action fires, component c is assigned to node n changing the value of the predicate *node_component* and n is marked as used.

Similarly the delete action frees the component assigned to a node. This action can be applied if component $c1$ is assigned to node n. The precondition also checks that all ports p provided by $c1$ and required by some other component $c2$, is provided also by a third component $c3$ (this ensure configuration correctness). As an effect it first frees node n. The code of for this action is given in Listing 7.5.

Listing 7.5: *del* action

```
1   (:action delete
2     :parameters (?n - node ?c1 - component)
3     :precondition (and
4       (node_component ?n ?c1)
5       (forall (?c2 - component ?n1 - node ?p - port)
6         (imply
7           (and
8               (node_component ?n1 ?c2)
9               (not (= ?n ?n1))
10              (component_requires_port ?c2 ?p)
11              (component_provides_port ?c1 ?p)
12          )
13          (exists (?c3 - component ?n2 - node)
14            (and
15              (node_component ?n2 ?c3)
16              (component_provides_port ?c3 ?p)
17              (not (= ?n ?n2))
18            )
19          )
20        )
21      )
22    )
23    :effect (and
24      (not (node_component ?n ?c1))
25      (not (used_node ?n))
26    )
```

```
27 )
```

Finally we present the code for the *stateChange* action in Listing 7.6. This action is encoded as replacing component $c1$ with component $c2$ on a given node n (where $c1$ is deployed). In order to enable this action some conditions must be met to guarantee configuration correctness. lines 7-24 deal with the verification that all ports provided by $c1$ and not by $c2$ are also provided by another component $c4$. lines 25-38 are, instead, used to verify that all required ports activated by $c2$ are provided by some other component $c3$.

Listing 7.6: *stateChange* action

```
1  (:action change_state
2    :parameters (?n - node ?c1 - component ?c2 - component)
3    :precondition (and
4      (node_component ?n ?c1)
5      (transition ?c1 ?c2)
6      ; requirements must be satisfied in the next state
7      (forall (?c3 - component ?n1 - node ?p - port)
8        (imply
9          (and
10           (not (= ?n ?n1))
11           (node_component ?n1 ?c3)
12           (component_requires_port ?c3 ?p)
13           (component_provides_port ?c1 ?p)
14           (not (component_provides_port ?c2 ?p))
15         )
16         (exists (?c4 - component ?n2 - node)
17           (and
18             (node_component ?n2 ?c4)
19             (component_provides_port ?c4 ?p)
20             (not (= ?n ?n2))
21           )
22         )
23       )
24     )
25     (forall (?p - port)
26       (imply
27         (and
28           (component_requires_port ?c2 ?p)
29         )
30         (exists (?c3 - component ?n1 - node)
31           (and
```

```
32                (node_component ?n1 ?c3)
33                (component_provides_port ?c3 ?p)
34                (not (= ?n ?n1))
35              )
36            )
37          )
38        )
39
40      )
41      :effect (and
42        (not (node_component ?n ?c1))
43        (node_component ?n ?c2)
44      )
45    )
```

Problem part

As for the problem instance it is specified by listing the objects, the initial state (as a set of predicates that hold) and the goal that one wants to achieve. A fragment of a sample problem file is given in Listing 7.7. First, we say which is the domain, *aeolus*. Then, is given a list of objects and a list of properties that hold (at the beginning). Finally, a property representing the goal is provided.

Listing 7.7: Excerpt from problem file

```
1  (:domain aeolus)
2  (:objects
3    node0 - node
4    node1 - node
5    node2 - node
6    comp_0_0 - component
7    comp_0_1 - component
8    comp_0_2 - component
9    ...
10   comp_1_0 - component
11   comp_1_1 - component
12   ...
13   port_0_1 - port
14   port_0_2 - port
15   ...
16 )
17 (:init
18   (initial_component comp_0_0)
```

```
19    (initial_component comp_1_0)
20    (transition comp_0_0 comp_0_1)
21    (transition comp_0_1 comp_0_2)
22    ...
23    (component_provides_port comp_0_1 port_0_1)
24    (component_provides_port comp_0_2 port_0_1)
25    (component_provides_port comp_0_2 port_0_2)
26  )
27  (:goal
28    (node_component node0 comp_1_3)
```

Test suite

As a benchmark we have considered Aeolus components automatically generated following the pattern of interdependency characterizing the running example. Moreover, for each test case we employed two classes of problem instances: first without and then with the need to apply the duplication procedure (Algorithm 6).

The first scenario, called Test A, and depicted in Figure 7.2, is designed to test "vertical scalability" as the number of states of the automaton increases. The scenario is composed by two components, C0 and C1, each of which has an automaton with n states. The goal is to reach the last state of component C1, labeled with n. To achieve this, the plan has to create the two components and to perform an alternating sequence of actions, namely a state change in component C0 followed by a state change in component C1 and the other way around, until the target state is reached. The interleaving of actions is enforced by means of patterns of required and provided ports.

The second scenario is, instead, designed to test "horizontal scalability", namely as the number of components increases. Usually in real life scenarios the number of states inside a component is rather small. We have thus considered configurations composed by an increasing number of components having all three states [2] as depicted in Figure 7.3. There are $n+1$ components, C0, C1, ..., Cn and the target is represented by state s2 of component Cn. The states of of the components' au-

[2]With exception of component C0, used to trigger the plan, having only two states.

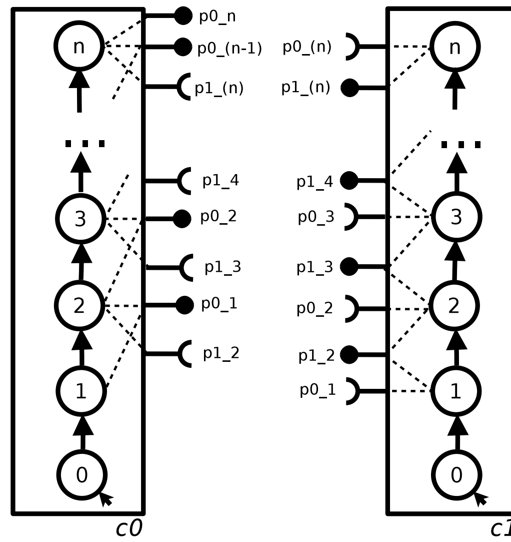


Figure 7.2: Experimental scenario for Test A.

tomaton activate provided and required ports in such a way that a valid plan must have the following form: first create all components, then perform from Cn to C0, in sequence, the state change from s0 to s1, and finally perform the state change from s1 to s2 from C1 to Cn.

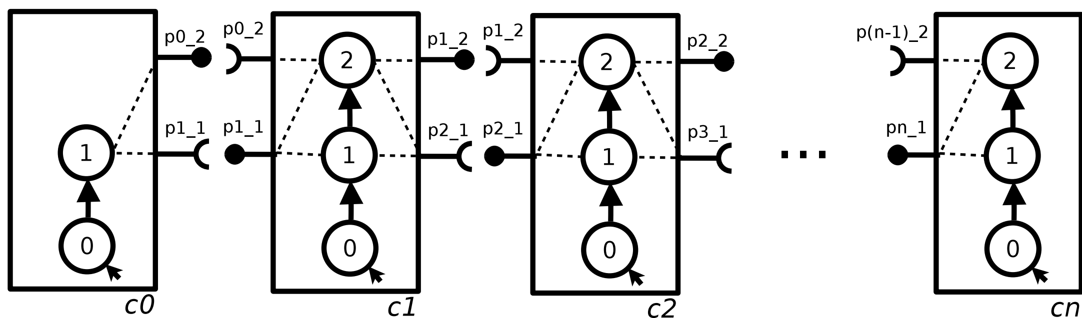


Figure 7.3: Experimental scenario for Test B.

Duplication

In these scenarios instance duplication is not needed during the generation of the concrete plan. As duplication may introduce a potentially significant computational overhead it must also be considered. To test this feature we modified the component types by randomly removing some provided ports in order to force duplication. We can accomplish this in TestA by deleting some of the provided ports from a state sX to the port $p0_(X-1)$ where X is the label of the state. Figure 7.4 shows red crossed arc corresponding to a possibly deleted provided port. Consider component c0. If port $p0_1$ is not provided by state 2, in order to reach it two instances of c0 are needed. A duplicate instance must stay in state 1 providing port $p0_1$ for state 1 of component c1, which, in its turn, is necessary for c0 to change state into 2.

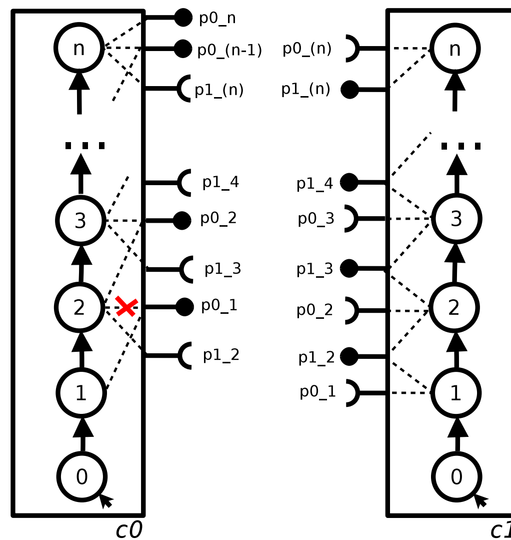


Figure 7.4: Modified Test A.

Similarly, for Test B we remove the activation of the provide port pX_1 from the states s2, where X is the number of the selected component. Removing this activation of the provided port demands duplicating the instance of component type CX in order to simultaneously satisfy the requirements of its two neighbor components. In Figure 7.5 is depicted the Test B scenario modified: the red crossed

arc corresponds to a provided port that might be removed. If the provided port *p1_1* is missing from state 2 of component c1, then if we want to reach it, a duplicate instance c1' must be created. c1' will be left into state 1 in order to keep providing port *p1_1* needed by state 1 of c0.

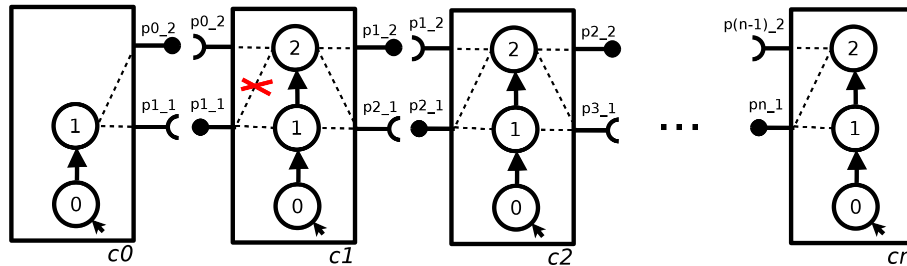


Figure 7.5: Modified Test B.

States and components affected by these deletions are chosen randomly. The number of deletions applied in both cases amounts to nearly one fifth of the total number of states, for Test A, and components, for Test B.

Experimental results

The tests were performed using a dual core machine with a 2.50 GHz Intel i5 processor, 6GB of RAM, and Ubuntu 12.10 operating system with 64 bit support.

We used a time cup of 130 seconds and two planners that support the ADL fragment of PDDL (other popular solvers support only fragments of PDDL): Metric-FF [10, 46] and Madagascar-p [6]. The first solver is based on GraphPlan [20], a standard planning algorithm to prune the search space, while the second, of the Satplan [51] family, encodes the planning problem into a SAT formula and then uses state of the art SAT solvers to find a solution. For reducing the search space of these solvers we set to the minimum the number of components that could be used concurrently.

We proceed in the following way: first, performance of the two regular planners is taken into account, then, we compare performance of METIS w.r.t. the planners and

finally, we evaluate performance of METIS in the basic and in the complex scenario (involving duplication).

Performance of the two planners are summarized in Table 7.1 and Table 7.2, dealing, respectively, with the A and B test case. Each table reports the time performances for both kinds of test in the basic and the complex scenario, denoted Test A+ and Test B+, where duplication is necessary.

We notice that the performances of the general purpose solvers are quite limited. Entry *error* indicates that the solver exited with an error state without computing the plan. Entry *timeout* means that the solver took more than 130s and was thus interrupted. The poor performances are due to the fact that the size of the encoding of the planning problem increases exponentially w.r.t. the number of components that need to be deployed concurrently. Metric-FF times out because it spends all the time trying to ground all the possible actions. Both Madagascar-p and Metric-FF terminate returning *error* because they exceed memory bound: the encoding into SAT for the former or the model containing all the ground atoms for the latter becomes too big to be handled.

Size	Test A		Test A+	
	Madagascar-p	Metric-FF	Madagascar-p	Metric-FF
5	0.10 s	0.01 s	0.17 s	0.20 s
10	0.97 s	0.13 s	6.92 s	timeout
15	5.10 s	0.49 s	error	error
25	error	2.53 s	–	–
35	–	7.98 s	–	–
45	–	20.27 s	–	–
55	–	47.97 s	–	–
65	–	error	–	–

Table 7.1: Performances of standard planners for Test A with and without duplication.

Let us analyze Table 7.1. For Test A case, Metric-FF has the best performance. It is able to deal with problem instances where the number of states does not go beyond 65. After that limit it fails eating up all the memory. For Test A+ case, both solvers get stuck quite early: Madagascar-p, performing slightly better, does not go beyond 15, while Metric-FF already at 10 times out and at 15 starts also to exceed memory capacity.

Size	Test B		Test B+	
	Madagascar-p	Metric-FF	Madagascar-p	Metric-FF
3	0.07 s	0.07 s	–	–
4	0.47 s	timeout	–	–
5	2.21 s	error	3.71 s	error
6	error	–	error	–

Table 7.2: Performances of standard planners for Test B with and without duplication.

Let us now turn attention to Table 7.2. In this test case the limitations are even more severe. The best planner among the two is Madagascar-p, but it is able to solve instances with only 5 components for both the basic Test B and Test B+ scenario. After that it starts to terminate with *error*. Metric-FF, already in the Test B case, times out even when considering a scenario with 4 components.

In the next graphs in Figure 7.6 and Figure 7.7 performance of METIS is compared w.r.t. the performance of both Metric-FF and Madagascar-p planners. Figure 7.6 takes into account the Test A scenario, while Figure 7.7 reports results for the Test B scenario. METIS outperforms the general purpose planners. In this phase it suffices to consider the basic scenario, without duplication, as performance of regular planners is so limited that, by lack of points, graphs would not show any particular trend in the complex scenario.

Let us look at Figure 7.6 first. As reported by Table 7.1 above, with regular planners the best results are obtained by Metric-FF, able to deal with instances

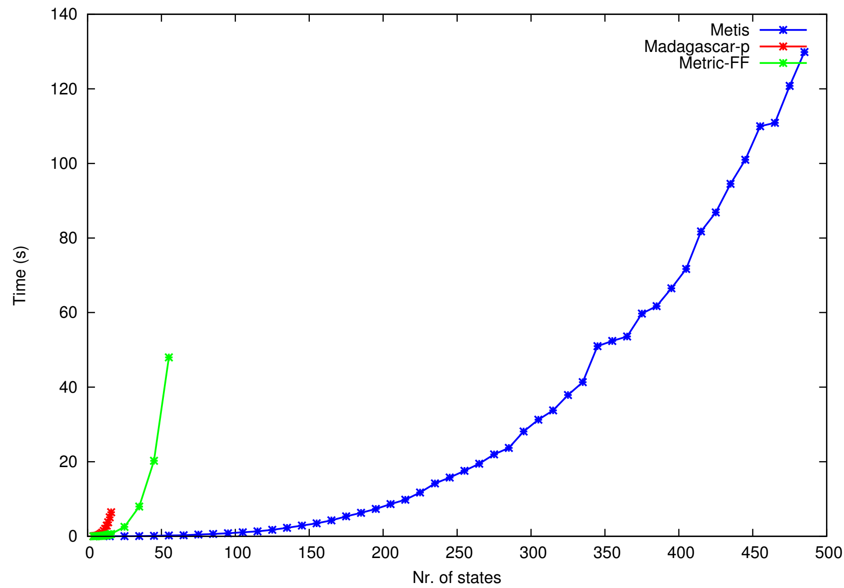


Figure 7.6: Performance comparison: METIS vs. regular planners on Test A.

with up to 65 states in nearly 48 s (in the same time METIS is able to deal with instances bigger than 335 states). METIS is able to synthesize a deployment plan for two components of up to 485 states in 129 s. After that it terminates having consumed all the available memory.

As for the Test B scenario, Figure 7.7 reports the following results. Madagascar-p and Metric-FF get stuck almost immediately: the former deals at most with 5 components while the latter terminates with error already at instances with 4 components. METIS succeeds in generating a deployment plan for instances of the problem with 225 components (of 3 states each) in nearly 65 s. After that eats up all the memory, returning error.

Notice that in this scenario, performance of METIS is apparently worse w.r.t. the Test A case. This, however, is no surprise if we consider the fact that Test B requires a growing number of concurrently active components while in Test A there are two components and, in principle, each of them is in only one of its states at every moment.

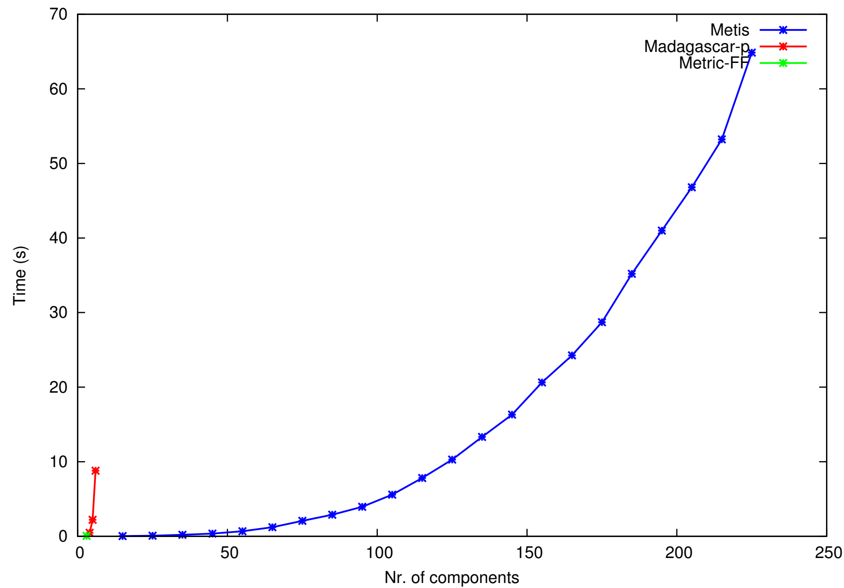


Figure 7.7: Performance comparison: METIS vs. regular planners on Test B.

Let us now examine the performance difference of METIS between the basic and the complex scenario. Results are depicted in Figure 7.8 and Figure 7.9, for the A and B test case, respectively. As expected, duplication affects the performance of the tool. From the graphs we can see that duplication does indeed add an overhead that one should take into account.

For the A test case the difference is quite significant. For the Test A+ scenario METIS is able to deal with instances of up to 155 states and afterwards it gets stuck because of memory limitations. This is mainly due to the fact that in this case, at the beginning, there are only two instances: one for each component, C0 and C1. Every time a duplication is performed all actions that the original instance performed must be copied and/or replaced by the duplicate instance. As the length of the instance lines grows with the size of the problem, so does the number of actions the need to be inserted and/or replaced (see lines 7-15 in Algorithm 6). This may translate into a heavy computational overhead as the deployment plan length increases asymptotically. Consider, for instance, the case of a problem instance of 100: for Test A+ the generated plan may involve around 3500 actions while for Test

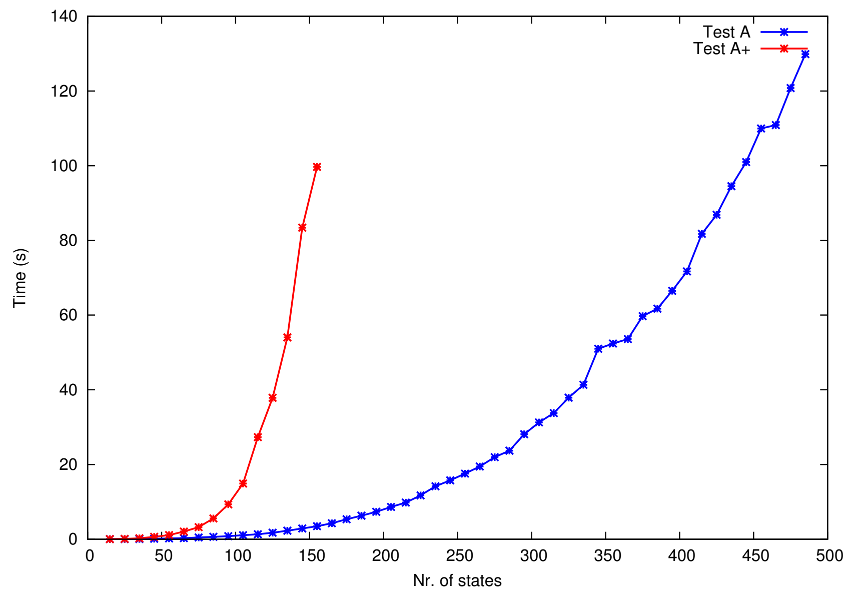


Figure 7.8: Performance of METIS on test Test A and Test A+.

B+ the plan obtained is about 800 actions long!

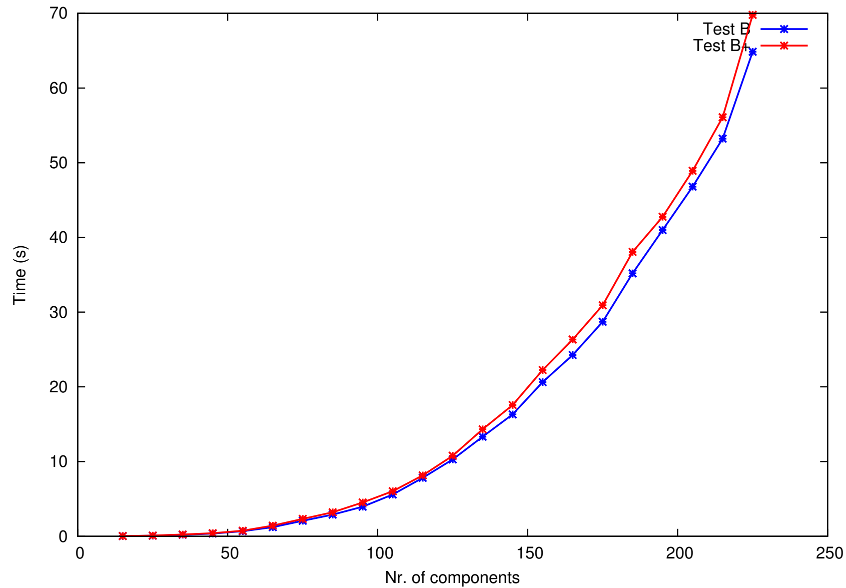


Figure 7.9: Performance of METIS on Test B and Test B+.

As for the other test case, scenario B, the performance degradation is rather

limited as testified by the closeness of the red graph, for Test B+ scenario, to the blue one, depicting the performance curve for the basic scenario Test B. In this case, the number of actions to be copied/replaced by duplication is much smaller as explained above. When duplication is required, METIS is nevertheless able to synthesize a deployment plan for more than 200 components in less than a minute.

It is important to stress the fact that, for the goal of this work, the B test case is somehow "more significant" than the A one, as the latter studies the situation for a growing number of states. In real life examples, however, the number of states is not expected to grow to an unbounded value (an automaton with 10-15 states is already a complex one). The number of components, on the other hand, rises naturally to high values in complex applications, involving many different components.

Part V

Conclusion

Chapter 8

Future directions

This chapter gives an overview of the planned extensions of the work presented so far. They are:

1. integration in the Aeolus toolchain;
2. introduction of a notion of conflict;
3. introduction of capacity constraints;
4. evaluation of alternative heuristics and their impact on the deployment plan;
5. dealing with full reconfiguration;
6. applying some form of restriction on the automata of the original Aeolus model.

Some of them have already been subject to discussion and preliminary work, others deserve further investigation. For the former we are able to provide few details, while for the latter we just give the intuition behind.

8.1 Integration in the Aeolus toolchain

One important step in the development of METIS is its integration in the Aeolus Deployment Engine, the toolchain built by the Aeolus team. In the end, this solution will leverage the scientific results produced in the Aeolus project to build

a full toolchain to be employed in the industrial setting of the industrial partner, Mandriva [7]. *Mandriva Business Server* (MBS) [8] is a full-fledged server solution supporting the deployment of enterprise applications. This is the setting chosen for industrial exploitation of the set of techniques and tools developed in the Aeolus project.

At the moment is already present a companion tool, called Zephyrus [12, 27], that enables one to compute an optimal final configuration starting from an initial one. The solution computed by Zephyrus is guaranteed to be optimal w.r.t. the number of (virtual) machines that need to be allocated to host all the running components. Zephyrus is able to start from an initial partial configuration to reach a final complete one, where all required components are listed, each one on its host machine, as well as the active bindings. Moreover, Zephyrus takes into account all the elements of the Aeolus flat model: capacity constraints as well as conflicts are dealt with. The final configuration fulfills all the capacity constraints and it does not contain any conflict, if such a solution exists. This achievement does not entail any contradiction with the work of presented in this dissertation as Zephyrus computes a final configuration that is the goal of the deployment process. The problem of finding a way, a plan, to actually achieve the computed configuration is out of its scope. This is due to the computability/decidability limitations inherent to the Aeolus model, outlined in Section 5.1.

The final toolchain envisioned by Aeolus should work in the following way: first an optimal final configuration $\mathcal{C}$ is computed by means of Zephyrus, then a deployment plan P is obtained by means of METIS and finally a tool, developed by Mandriva, translates the actions specified by P into the installation/activation/provisioning steps, specific to the MBS environment.

Two issues arise naturally: conflicts and capacity constraints. As mentioned above the solution computed by Zephyrus takes them into account while METIS does not: hence there is a gap that needs to be taken care of. For conflicts, the idea is that if violations occur during deployment, in a transient configuration, this may still be manageable, as long as the final configuration is conflict-free and possible violations

are detected/signaled in advance. In this situation the system administrator might take care of the problem by hands. METIS might be extended to signal a potential conflict whenever two conflicting states might be concurrently active. This can be established by analyzing the abstract plan: if the topological order does not ensure that the conflicting states are reached and left one before the other, then a potential violation is signaled.
For capacity constraints, one might say that the final configuration $\mathcal{C}$ computed by Zephyrus is a maximal one, that is to say: the deployment plan will never need more instances of a given component than the number present in $\mathcal{C}$, nor will it require a larger set of components. This is a reasonable assumption as may be ensured by adopting a monotonicity principle: if a state s requires a given port r and provides some set of ports P, then a successor state s', providing more functionalities ($P \subseteq P'$), must also activate such a requirement.

Notice that these ideas apply to the current version of METIS. Some possible enhancements enabling it to deal with conflicts and capacity constraints at a more natural level, are outlined in Section 8.2 and Section 8.3 below.

Finally, we would like to underline that the above ideas abstract away from the target industrial application in mind and as such may be applied in a general platform independent approach. Such a development may, for instance, exploit existing tools for configuration management, mapping the kind of actions used in our deployment plans into platform independent languages, such as Puppet.

8.2 Conflicts

The Aeolus^{-} model, considered in this thesis, abstracts from the notion of *conflict* among components. Conflicting components arise naturally at the level of packages in Linux-like distributions, where the presence of two packages at the same time on the same machine may be not admissible. At the level of services, in a distributed setting, there are some (few) examples of conflicts, the most common of which being that of two DNS servers on the same network. As such, conflicts are part of the

original Aeolus model. The kind of conflicts considered is a "global" one: a conflict has global scope, ranging on all components of a given configuration. Dealing with conflicts is a hard problem as testified by the fact that, to our knowledge, no existing approach/tool takes them into account. In fact, the problem has been formally proven to be in EXPSPACE [28]. [1]

One of the development directions that we are currently exploring is trying to integrate conflicts into our approach. In this "enhanced version" METIS tries to avoid conflicts whenever possible and if it does not succeed it raises a warning, signaling a potential conflict violation during deployment. To understand the ideas underlying this integration one needs to take a slightly different point of view on the concept of plan. The plan P generated by METIS is a sequence of actions. Moreover, this sequence is a linearly ordered one that is the result of performing the adaptive topological sort described in Section 6.2.3. In Algorithm 5 duplication is carried out while visiting the nodes of the abstract plan $\mathsf{P}_{\#}$. One could also, on the other hand, perform first duplication and visit the nodes only afterwards, on a "modified" abstract plan ${\mathsf{P}_{\#}}^{+}$. Namely, ${\mathsf{P}_{\#}}^{+}$ is obtained by performing duplication on the original abstract plan $\mathsf{P}_{\#}$. [2] Figure 6.12 represents an example of such a ${\mathsf{P}_{\#}}^{+}$ and can be compared to the original abstract plan $\mathsf{P}_{\#}$ depicted in Figure 7.1.

To deal with conflicts one may work at the ${\mathsf{P}_{\#}}^{+}$ plan level. The idea is to introduce a new type of arrow that represents precedence constraints related to conflicts, much in the same way as $\dashrightarrow$ and $\twoheadrightarrow$ are used to model action precedence relative to provided and required interfaces. An example is shown in Figure 8.1 below.

This picture shows a sample of the ${\mathsf{P}_{\#}}^{+}$ plan in a scenario involving two conflicting instances: i of component type $\mathcal{I}$ and m of component type $\mathcal{M}$. State y of $\mathcal{I}$ is in conflict w.r.t. state q of $\mathcal{M}$. The pair of arcs 1 and 2 express the fact that

[1] Effective technique and tools to deal with conflicts at the level of packages on a single machine, have been devised in [13, 57]. The Aeolus context corresponds, however, to an "upgraded" one, in a distributed setting.

[2] Notice that performing duplication as a first step implies the explicit use of cycle detection techniques in order to find the instances that need to be duplicated.

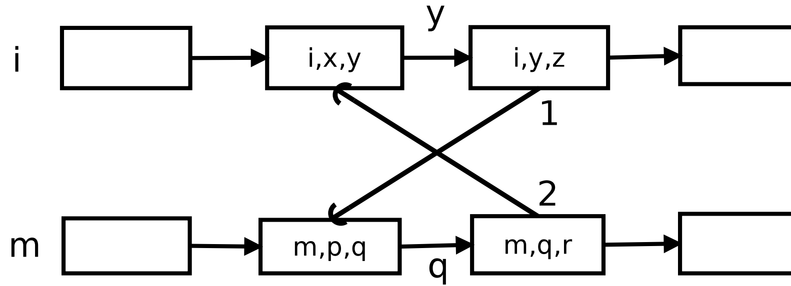


Figure 8.1: Conflict detection information on abstract plan.

in a deployment plan the conflict is avoided if either:

- instance i leaves state y before m enters state q (arc 1: node $\langle i, y, z \rangle$ is visited before $\langle m, p, q \rangle$), or
- instance m leaves state q before i enters state y (arc 2: node $\langle m, q, r \rangle$ is visited before $\langle i, x, y \rangle$).

It is important to stress two aspects concerning conflict arcs. First, they are mutually exclusive: if precedence specified by 1 is respected, then precedence specified by 2 does not hold. Visually, when taking into account arc 1, the other twin arc, 2, is canceled. Second, the kind of constraints specified by conflict arcs differs from the one specified by the other arcs in the abstract plan. The order imposed by the latter must be respected in order to find a plan that is correct; the order imposed by the former, instead, may be interpreted as a "preference": if possible i should leave state y before m enters state q (or the other clause). If this is not possible, a conflict may arise during deployment.

Notice that, by construction, the $\mathsf{P_\#}^+$ graph is free of circularities involving only $\longrightarrow$, $\dashrightarrow$ and $\twoheadrightarrow$ arcs (as duplication has already broken any existing such cycle). Circularities in $\mathsf{P_\#}^+$ may show up if we consider the new kind of conflict-arc.

The idea is to exploit techniques similar to the ones used in the basic setting (without conflicts). If cycles appear in the $\mathsf{P_\#}^+$ augmented with conflict-arcs, in the enhanced version, METIS would try to break it by removing one among two twin

conflict-arcs. A sound technique would involve, in the worst case scenario, analyzing all possible combinations of pairs of conflict-arcs to obtain a cycle-free plan. This, however, would lead to an exponential explosion in the complexity of the algorithm. An alternative possibility that seems viable is, instead, trying to break cycles by choosing and removing some conflict-arcs. This choice might be performed relying on some heuristics, as done in the original technique. If the combination of choices achieves cycle-freedom, the deployment plan is ensured to be conflict-free, otherwise for every arc removal that is not able to break a cycle, a warning is issued saying which components' states can potentially be in conflict during deployment.

8.3 Capacity constraints

The idea to extend our approach to deal with capacity constraints is that one may take them into account during component selection, while performing the bottom-up visit (Section 6.2.2). For every selected node of a given level, one must choose an origin and a provider for every required interface p. Now imagine that for a node $\langle \mathcal{T}, q \rangle$ to required port p is associated a capacity value of n: this means that for p to be fulfilled, it needs a port with multiplicity at least equal to n. If a provider $\langle \mathcal{T}', q' \rangle$ has capacity equal to m, associated to port p, it means it can serve up-to m required ports simultaneously. Then, we know for sure that in order to fulfill node $\langle \mathcal{T}, q \rangle$'s requirements for port p, we need at least $k = \lceil \frac{n}{m} \rceil$ instances of type $\mathcal{T}'$. So now we would need to take also this new information into account: a numeric value must be assigned to every arc; we must keep track of the number of instances of a given type and the "amount of usage" of each provided port [3]. It is not obvious if it is better to store this information at the reachability graph level during selection (like in Figure 6.8) or at the abstract plan level (for example on the $\twoheadrightarrow$ arcs). Abstract planning starts with a single instance per component type and additional instances, if needed, are created as effect of the duplication process. One

[3] Of course if there there are some ports not fully used in bindings one could try to reuse them before asking for more instances.

should change accordingly the abstract planning procedure. Moreover, the interplay between duplication and capacity constraints must be studied. Imagine an instance x, working as a provider of port p with capacity of 4, is duplicated, then the total amount of p provided should be subdivided among the two instances x and x'. One could say that x participates with 2 and so does x'.

Overall it looks like a feasible extension of the current work, even though not trivial [4], as it is a matter of adapting the already existing technique.

8.4 Heuristics

The heuristics currently employed by METIS and defined in Section 6.2.4 represent a tentative version to accomplish a deployment plan exploiting a minimal number of components. As already mentioned, attaining the least number of components corresponds to finding a global minimum value in the presence of disjunction of possibilities (namely different predecessors and different providers for the same node and requirement). Several variations of the adopted strategy must be examined. One could, for example, give preference to a copy node rather than to the node with minimum cardinality value (currently it is the other way around). In order to assess the advantages/drawbacks exposed by the possible variations we need to devise a suite of tests based on different scenarios.

8.5 Reconfigurations

Up until now the work is based on the underlying assumption that the starting/initial configuration is empty. This means that the deployment problem is dealing with a deployment "from scratch". In the real world, however, most of the times the system is already up and running and one needs to apply a *reconfiguration* rather than a full deployment. Dealing with reconfigurations is not a trivial task as they usually

[4] The (component selection) heuristics too have to be adapted.

involve also the deletion of components in use. The possibility of deleting components introduces the possibility of producing configurations that are not correct: a requirement may be fulfilled by a component that suddenly disappears.

As explained in Section 6.2.1 the reachability analysis part works *forward*, by saturation. This approach is heavily based on the monotonicity property of the set of the components that become deployable at every step. There is no possibility to invalidate a configuration generated at a given level of the reachability graph because we are always adding newly available/reachable components. If we abandon this assumption a major change in our framework is needed, left to future research.

8.6 Restrictions

As already noticed, the negative results summarized in Section 5.1 hold in the general case where automata are allowed to have arbitrary complexity: any number and configuration of states, required/provided ports. In real life scenarios, however, it is unlikely for components to exhibit such complex behaviour. A still open question is thus whether it is possible to devise an efficient solution by imposing some form of limitation on the automaton structure of the original Aeolus model. Tentative restrictions that have been proposed are the following:

1. fix the maximum (possibly small) number of states;
2. monotonicity of states w.r.t. the provided/required ports. If a state has some required and some provided ports, then its successor(s) should activate at least the same ports and possibly more.

The effects of these (or possibly other) restrictions on the feasibility/complexity of the deployment problem deserve further investigation.

Chapter 9

Concluding remarks

This dissertation describes the design and implementation of a technique for the automatic synthesis of deployment plans. A direct translation of the deployment problem into PDDL, a format suitable for standard planning tools, has been showed to lead to an unfeasible solution. Thus, a novel approach, based on meaningful component abstractions, has been carefully devised in order to efficiently cope with problem instances involving a high number of components. The validity of the proposed technique is supported by formal results that prove its soundness and completeness. Moreover, the central algorithm is shown to have computational complexity that is polynomial w.r.t. the number of available components. Finally, viability of the technique in practice is witnessed by means of experimental results that exhibit encouraging performances.

List of Algorithms

1 *DeploymentPlanner* pseudocode 57
2 Reachability graph construction 62
3 Component Selection 67
4 Abstract Plan Generation 71
5 Plan synthesis 75
6 Duplication 79

List of Tables

3.1 Available techniques & tools for deployment automation. 25

5.1 Results for the achievability problem. 44

6.1 Kind of temporal precedence arcs. 69

6.2 Parameters' precedence order for the heuristics employed. 82

7.1 Performances of standard planners for Test A with and without duplication. 107

7.2 Performances of standard planners for Test B with and without duplication. 108

List of Figures

3.1 Example of hypergraph generated by Engage. 12
3.2 Typical state machine associated to a resource driver. 13
3.3 ConfSolve workflow. 14

5.1 Typical Wordpress/Apache/MySQL deployment, modeled in Aeolus. 36
5.2 A graphical description of the model with redundancy and capacity constraints (internal sate machines and activation arcs omitted for simplicity). 37
5.3 On the need of a *multiple state change* action: how to install a and b? 42
5.4 Components employed in the encodings. 44

6.1 Chain of three phases. 56
6.2 MySQL master-slave components according to the Aeolus model . . . 58
6.3 MySQL master-slave replication final configuration 59
6.4 Sample deployment plan for the running example (part 1). 60
6.5 Sample deployment plan for the running example (part 2). 61
6.6 Reachability graph for the running example. 63
6.7 Difference between lazy and complete strategies. 65
6.8 Component selection for the running example. 66
6.9 Abstract plan for the running example. 70
6.10 Initial configuration for the new scenario. 77
6.11 Cyclic abstract plan for the modified running example. 78
6.12 Abstract plan after duplication. 78
6.13 Basic scenario (only 3 arcs). 86

7.1 Abstract plan output for the running example. 97
7.2 Experimental scenario for Test A. 104
7.3 Experimental scenario for Test B. 104
7.4 Modified Test A. 105
7.5 Modified Test B. 106
7.6 Performance comparison: METIS vs. regular planners on Test A. 109
7.7 Performance comparison: METIS vs. regular planners on Test B. 110
7.8 Performance of METIS on test Test A and Test A+. 111
7.9 Performance of METIS on Test B and Test B+. 111

8.1 Conflict detection information on abstract plan. 119

References

[1] Aeolus: Mastering the Complexity of the Cloud. http://www.aeolus-project.org/. Project ANR-2010-SEGI-013-01.

[2] CFEngine. http://cfengine.com/.

[3] DevOps. http://devops.com/.

[4] Google App Engine. https://developers.google.com/appengine/.

[5] JSON. http://www.json.org/.

[6] Madagascar-p. http://users.ics.aalto.fi/rintanen/jussi/satplan.html.

[7] Mandriva SA. www.mandriva.com.

[8] MBS - Mandriva Business Server. http://www.mandriva.com/en/products-services/mbs/.

[9] METIS - Modern Engineered Tool for Installing Software systems. https://github.com/aeolus-project/metis.

[10] Metric-FF. http://fai.cs.uni-saarland.de/hoffmann/metric-ff.html.

[11] Windows Azure. http://www.windowsazure.com.

[12] Zephyrus. https://github.com/aeolus-project/zephyrus.

[13] Pietro Abate, Roberto Di Cosmo, Ralf Treinen, and Stefano Zacchiroli. MPM: a modular package manager. In Ivica Crnkovic, Judith A. Stafford, Antonia Bertolino, and Kendra M. L. Cooper, editors, *CBSE*, pages 179–188. ACM, 2011.

[14] Franz Achermann and Oscar Nierstrasz. A Calculus for Reasoning about Software Composition. *Theor. Comput. Sci.*, 331(2-3):367–396, 2005.

[15] Aeolus by RedHat. http://www.aeolusproject.org/.

[16] N. Arshad, D. Heimbigner, and A.L. Wolf. Deployment and Dynamic Reconfiguration Planning for Distributed Software Systems. In *Tools with Artificial Intelligence, 2003. Proceedings. 15th IEEE International Conference on*, pages 39–46, Nov 2003.

[17] Christer Bäckström and Bernhard Nebel. Complexity results for SAS+ planning. *Computational Intelligence*, 11(4):625–655, 1995.

[18] C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.

[19] T. Binz, G. Breiter, F. Leyman, and T. Spatzier. Portable Cloud Services Using TOSCA. *Internet Computing, IEEE*, 16(3):80–85, 2012.

[20] Avrim L. Blum and Merrick L. Furst. Fast Planning Through Planning Graph Analysis. *Artificial Intelligence*, 90(1–2):281 – 300, 1997.

[21] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The FRACTAL Component Model and its Support in Java. *Softw., Pract. Exper. '06*, 36(11-12):1257–1284, 2006.

[22] Mikkel Bundgaard, Thomas T. Hildebrandt, and Jens Chr. Godskesen. A CPS Encoding of Name-Passing in Higher-Order Mobile Embedded Resources. *Theor. Comput. Sci.*, 356(3):422–439, 2006.

[23] Mark Burgess. A Site Configuration Engine. *Computing Systems*, 8(2):309–337, 1995.

[24] Tom Bylander. Complexity Results for Planning. In *IJCAI*, volume 10, pages 274–279, 1991.

[25] CompatibleOne. www.compatibleone.org.

[26] Luca De Alfaro and Thomas A Henzinger. Interface Automata. In *ACM SIGSOFT Software Engineering Notes*, volume 26, pages 109–120. ACM, 2001.

[27] Roberto Di Cosmo, Michael Lienhardt, Ralf Treinen, Stefano Zacchiroli, and Jakub Zwolakowski. Optimal Provisioning in the Cloud. Technical report, Aeolus project, June 2013. http://hal.archives-ouvertes.fr/hal-00831455.

[28] Roberto Di Cosmo, Jacopo Mauro, Stefano Zacchiroli, and Gianluigi Zavattaro. Component Reconfiguration in the Presence of Conflicts. In *ICALP 2013: 40th International Colloquium on Automata, Languages and Programming*, volume 7966 of *LNCS*, pages 187–198. Springer-Verlag, 2013.

[29] Roberto Di Cosmo, Jacopo Mauro, Stefano Zacchiroli, and Gianluigi Zavattaro. Component Reconfiguration in the Presence of Conflicts. Technical report, Aeolus Project, 2013. http://hal.archives-ouvertes.fr/hal-00816468.

[30] Roberto Di Cosmo, Stefano Zacchiroli, and Gianluigi Zavattaro. Towards a Formal Component Model for the Cloud. In *SEFM 2012: 10th International Conference on Software Engineering and Formal Methods*, volume 7504 of *LNCS*, pages 156–171. Springer-Verlag, 2012.

[31] DMTF (Distributed Management Task Force). Open Virtualization Format Specification Version 2.0.1. http://dmtf.org/sites/default/files/standards/documents/DSP0243_2.0.1.pdf.

[32] Kutluhan Erol, Dana S Nau, and Venkatramana S Subrahmanian. Complexity, decidability and undecidability results for domain-independent planning. *Artificial Intelligence*, 76(1):75–88, 1995.

[33] Etchevers, X. and Coupaye, T. and Boyer, F. and de Palma, N. Self-Configuration of Distributed Applications in the Cloud. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 668–675, July 2011.

[34] 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*, PLDI '12, pages 263–274, New York, NY, USA, 2012. ACM.

[35] Maria Fox and Derek Long. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *J. Artif. Intell. Res. (JAIR)*, 20:61–124, 2003.

[36] Gecode: Genetic constraint development environment. http://www.gecode.org.

[37] Michael Gelfond and Vladimir Lifschitz. Action Languages. *Electronic Transactions on AI*, 3(16), 1998.

[38] Alfonso Gerevini, Patrik Haslum, Derek Long, Alessandro Saetti, and Yannis Dimopoulos. Deterministic planning in the fifth international planning competition: PDDL3 and experimental evaluation of the planners. *Artif. Intell.*, 173(5-6):619–668, 2009.

[39] Alfonso Gerevini and Ivan Serina. LPG: A Planner Based on Local Search for Planning Graphs with Action Costs. In *AIPS*, volume 2, pages 281–290, 2002.

[40] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated planning: theory & practice*. Morgan Kaufmann, 2004.

[41] Patrick Goldsack, Julio Guijarro, Steve Loughran, Alistair Coles, Andrew Farrell, Antonio Lain, Paul Murray, and Peter Toft. The SmartFrog Configuration Management Framework. *SIGOPS Oper. Syst. Rev.*, 43(1):16–25, January 2009.

[42] Graphviz - graph visualization software. http://www.graphviz.org/.

[43] David Harel and P.S. Thiagarajan. Message Sequence Charts. In *UML for Real*, pages 77–105. Springer, 2004.

[44] John A. Hewson and Paul Anderson. Modelling System Administration Problems with CSPs. *Constraint Modelling and Reformulation (ModRef'11)*, 2011.

[45] John A. Hewson, Paul Anderson, and Andrew D. Gordon. A Declarative Approach to Automated Configuration. In *Proceedings of the 26th international conference on Large Installation System Administration: strategies, tools, and techniques*, pages 51–66. USENIX Association, 2012.

[46] Jörg Hoffmann. The Metric-FF planning system: translating "Ignoring delete lists" to numeric state variables. *J. Artif. Int. Res.*, 20(1):291–341, December 2003.

[47] Juraj Hromkovic and Waldyr M. Oliva. *Algorithmics for Hard Problems*. Springer-Verlag New York, Inc., 2nd edition, 2002.

[48] IBM. SmartCloud Orchestrator. http://www-03.ibm.com/software/products/us/en/smartcloud-orchestrator/.

[49] Juju, DevOps distilled. https://juju.ubuntu.com/.

[50] Luke Kanies. Puppet: Next-generation configuration management. *;login: the USENIX magazine*, 31(1):19–25, 2006.

[51] Henry A. Kautz, Bart Selman, et al. Planning as Satisfiability. In *ECAI*, volume 92, pages 359–363, 1992.

[52] Eleftherios Koutsofios, Stephen North, et al. Drawing graphs with dot. Technical report, Technical Report 910904-59113-08TM, AT&T Bell Laboratories, Murray Hill, NJ, 1991.

[53] Tudor Alexandru Lascu, Jacopo Mauro, and Gianluigi Zavattaro. A Planning Tool Supporting the Deployment of Cloud Applications. In *IEEE 25th International Conference on Tools with Artificial Intelligence, ICTAI 2013*, pages 213–220, 2013. http://lascu.web.cs.unibo.it/documents/ictai2013-lmz.pdf.

[54] Tudor Alexandru Lascu, Jacopo Mauro, and Gianluigi Zavattaro. A Planning Tool Supporting the Deployment of Cloud Applications. Technical report, Aeolus ANR Project, June 2013. http://hal.inria.fr/hal-00843925.

[55] Tudor Alexandru Lascu, Jacopo Mauro, and Gianluigi Zavattaro. Automatic Component Deployment in the Presence of Circular Dependencies. In *10th International Symposium on Formal Aspects of Component Software, FACS 2013*, 2013. http://lascu.web.cs.unibo.it/documents/facs2013-lmz.pdf.

[56] Yu David Liu and Scott F. Smith. A Formal Framework for Component Deployment. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 325–344, New York, NY, USA, 2006. ACM.

[57] Fabio Mancinelli, Jaap Boender, Roberto Di Cosmo, Jerome Vouillon, Berke Durak, Xavier Leroy, and Ralf Treinen. Managing the Complexity of Large Free and Open Source Package-Based Software Distributions. In *ASE*, pages 199–208, 2006.

[58] Drew V. McDermott. The 1998 AI Planning Systems Competition. *AI Magazine*, 21(2):35–55, 2000.

[59] Marvin Minsky. *Computation: finite and infinite machines*. Prentice Hall, 1967.

[60] Jelena Mirkovic, Ted Faber, Paul Hsieh, Ganesan Malaiyandisamy, and Rashi Malaviya. DADL: Distributed Application Description Language. *USC/ISI Technical Report# ISI-TR-664*, 2010.

[61] Fabrizio Montesi and Davide Sangiorgi. A Model of Evolvable Components. In *Trustworthy Global Computing*, volume 6084 of *Lecture Notes in Computer Science*, pages 153–171. Springer, 2010.

[62] Nicholas Nethercote, PeterJ. Stuckey, Ralph Becket, Sebastian Brand, GregoryJ. Duck, and Guido Tack. MiniZinc: Towards a standard CP modelling language. In Christian Bessière, editor, *Principles and Practice of Constraint Programming – CP 2007*, volume 4741 of *Lecture Notes in Computer Science*, pages 529–543. Springer Berlin Heidelberg, 2007.

[63] OASIS. Organization for the Advancement of Structured Information Standards (OASIS). https://www.oasis-open.org.

[64] OASIS. Topology and Orchestration Specification for Cloud Applications (TOSCA) Version 1.0. http://docs.oasis-open.org/tosca/TOSCA/v1.0/cs01/TOSCA-v1.0-cs01.html.

[65] OpenNebula. http://www.opennebula.org/.

[66] OpenStack. http://www.openstack.org.

[67] M.P. Papazoglou and W. van den Heuvel. Blueprinting the Cloud. *Internet Computing, IEEE*, 15(6):74–79, 2011.

[68] C. A. Petri. *Kommunikation mit Automaten, PhD thesis.* Institut für Instrumentelle Mathematik, Bonn, Germany, 1962.

[69] Puppetlabs. Puppet. http://puppetlabs.com/.

[70] C. Rackoff. The covering and boundedness problems for vector addition systems. *Theoret. Comp. Sci.*, 6:223–231, 1978.

[71] Gwen Salaün, Xavier Etchevers, Noel Palma, Fabienne Boyer, and Thierry Coupaye. Verification of a Self-configuration Protocol for Distributed Applications in the Cloud. In Javier Cámara, Rogério Lemos, Carlo Ghezzi, and Antónia Lopes, editors, *Assurances for Self-Adaptive Systems*, volume 7740 of *Lecture Notes in Computer Science*, pages 60–79. Springer Berlin Heidelberg, 2013.

[72] Davide Sangiorgi and David Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge university press, 2003.

[73] Alan Schmitt and Jean-Bernard Stefani. The Kell Calculus: A Family of Higher-Order Distributed Process Calculi. In *Global Computing*, volume 3267 of *LNCS*, pages 146–178. Springer, 2004.

[74] Lionel Seinturier, Philippe Merle, Damien Fournier, Nicolas Dolet, Valerio Schiavoni, and Jean-Bernard Stefani. Reconfigurable SCA Applications with the FraSCAti Platform. In *IEEE SCC*, pages 268–275. IEEE, 2009.

[75] VMWare. Cloud Foundry. http://www.cloudfoundry.com.

[76] Johannes Wettinger, Vasilios Andrikopoulos, Steve Strauch, and Frank Leymann. Enabling Dynamic Deployment of Cloud Applications Using a Modular and Extensible PaaS Environment. In *Cloud Computing (CLOUD), 2013 IEEE Sixth International Conference on*, pages 478–485. IEEE, 2013.

All the cited websites have been visited in December 2013.