Automated Planning for Cloud Service Configurations

Herry

Doctor of Philosophy

Centre for Intelligent Systems and their Applications
School of Informatics
University of Edinburgh

2015

Abstract

The declarative approach has been widely accepted as an appropriate way to manage configurations of large scale systems – the administrators describe the specification of the "desired" configuration state of the system, and the tool computes and executes the necessary actions to bring the system from its current state into this desired state. However, none of state-of-the-art declarative configuration tools make any guarantees about the order of the changes across the system involved in implementing configuration changes.

This thesis presents a technique that addresses this issue – it uses the SFP language to allow administrators to specify the desired configuration state and the global constraints of the system, compiles the specified reconfiguration task into a classical planning problem, and then uses an automated planning technique to automatically generate the workflow. The execution of the workflow can bring the system into the desired state, while preserving the global constraints during configuration changes.

This thesis also presents an alternative approach to deploy the configurations – the workflow is used to automatically choreograph a set of reactive agents which are capable to autonomously reconfigure a computing system into a specified desired state. The agent interactions are guaranteed to be deadlock/livelock free, can preserve prespecified global constraints during their execution, and automatically maintain the desired state once it has been achieved (*self-healing*).

We present the formal semantics of SFP language, the technique that compiles SFP reconfiguration tasks to classical planning problems, and the algorithms for automatic generation and execution of the reactive agent models. In addition, we also present the formal semantics of core subset of SmartFrog language which is the foundation of SFP. Moreover, we present a domain-independent technique to compile a planning problem with extended goals into a classical planning problem.

As a proof of concept, the techniques have been implemented in a prototype configuration tool called Nuri, which has been used to configure typical use-cases in cloud environment. The experiment results demonstrate that the Nuri is capable of planning and deploying the configurations in a reasonable time, with guaranteed constraints on the system throughout reconfiguration process.

Acknowledgements

The greatest thanks go to Mr. Paul Anderson, Dr. Michael Rovatsos and Dr. Gerhard Wickler who accepted me as a student, and then supervised me during my study. They gave many useful advices and other supports that helped me during difficult times in Edinburgh. Their openness and friendliness made our discussions to be very productive.

This thesis will not exist without encouragement from my beloved wife, Vicky Vilsy, to apply the scholarship. She also has been very patience and supportive during my study. I am also very grateful to my parents in Jakarta, Indonesia, who supported me in many things.

Many thanks to Lawrence Wilcock, Eric Deliot, Julio Guijarro and other researchers from Hewlett-Packard Laboratory, who have spent their times discussing various things about my research, gave me useful advices, and also made my internship in Bristol very pleasant. Special thanks to Patrick Goldsack who has taught me many things about SmartFrog.

I am also in debt to Dr. James Cheney who has given me useful advices about Denotational Semantics, and provided some extra fundings for my study. I would also thank to Andrew Farrell who has taught me about Behavioural Signatures model and other things during early days of my study. Thanks to John Hewson who has been given valuable feedbacks. And thanks to Prof. Heru Suhartanto and Dr. M. Rahmat Widyanto from the Universitas Indonesia, who have given me recommendation letters for my application.

My study at the University of Edinburgh was fully funded by a grant from HP Labs Innovation Research Program award.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Herry*)

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Cloud computing is now an accepted option for deploying services – services can be easily created, deleted and relocated to cater for changes in requirements, demands and economics. These capabilities are supported by large scale computing infrastructures whose size and complexity are growing inevitably. On the other side, configuring and managing cloud services are not easy tasks; the cloud environment is highly distributed and federated with unpredictable loading; and there is a wide variety of configuration aspects such as network, application service, and security, each of which may have dependencies on others. These increase awareness of the need for good system configuration tools in order to fully exploit the potential of the cloud, and most sites now use some form of tool to manage their configurations.

From various approaches, *declarative* specifications are now widely accepted as the most appropriate one – the configuration specification describes the "desired" state of the system, and the tool computes and executes the necessary actions to bring the system from its current state into this desired state. This has an advantage that the final *state* of the system is explicitly specified. Thus, we can have a confidence that the state of the system matches with our requirements. Other approaches are more error-prone because they involved specifying the *actions* (for example, imperative scripts), where the final outcome would not always be obvious. With varying degrees of strictness, most of the current popular tools take a broadly declarative approach, for example Puppet [Puppet Labs, 2014], LCFG [Anderson and Scobie, 2002], and BCFG [Desai et al., 2003].

However, none of the above tools make any guarantees about the order of the changes across the system involved in implementing a configuration change. This is not normally an issue when deploying a new service – we define the specification

and the tool implements the necessary changes in some "random" order. When it has finished, we have a running system matching to our specification. But, if we are making a configuration change (reconfiguration) to an existing system, we may well care about the intermediate states of the configuration during the transition. For example, if we want to transition from using one to another server, then we probably want to start the new server first, and transfer the clients before shutting down the old one.

One approach to the above problem has been the use of provisioning tools – we define and store static workflows so they can be invoked and scheduled automatically by a central controller to satisfy the ordering constraints. IBM Tivoli [IBM Corp., 2014], Microsoft System Center [Microsoft Corp., 2014], Ansible [Ansible Inc., 2014], and Chef [Opscode Inc., 2014] are the example tools which provide this kind of capability. However, this still requires the workflows to be computed manually. Even in a small system, a very large number of workflows would be required to cater for every eventuality. In addition, choosing an appropriate workflow to suit to a particular desired state is not always obvious – this is conceptually similar to the imperative scripts which are no longer popular because of their unreliability.

An alternative approach is to make use of automated planning techniques which generate workflows "on the fly". This allows us to define a specification that consists

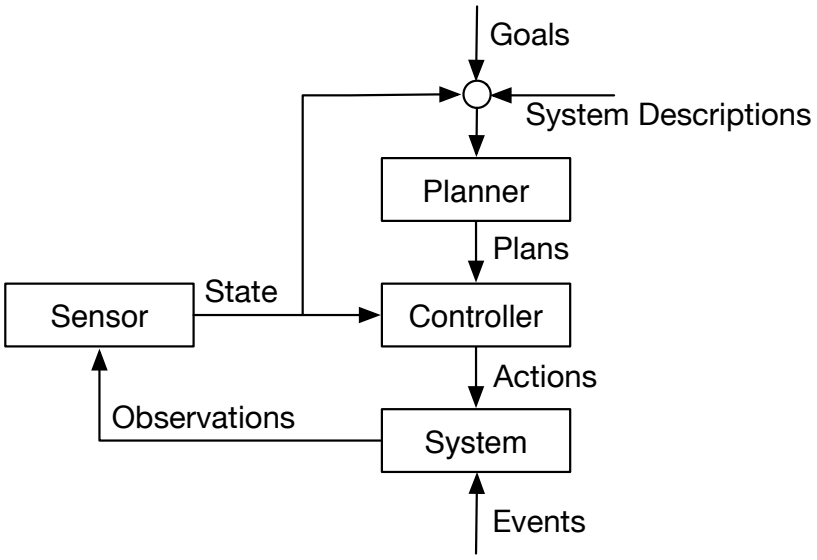


Figure 1.1: A conceptual architecture of configuration tools that make use of automated planner.

of the current state[1] as well as the desired state, together with a set of constraints defined in actions. Based on this specification, the planner automatically generates the workflow which is then executed to implement the transition of the system between two arbitrary states without violating the specified constraints. This has an advantage that the tool can be used either for deploying a new system from scratch or for making a configuration change. Figure 1.1[2] illustrates a common conceptual architecture of system configuration tools which make use of automated planner. [Keller et al., 2004, El Maghraoui et al., 2006, Levanti and Ranganathan, 2009, Hagen and Kemper, 2010] are some previous works which are using this kind of approach.

However, none of these works allow us to define the *global constraints* – a set of invariant constraints that must be satisfied at the intermediate and final states, as the goals[3] of the system. All constraints must be defined explicitly as preconditions associated with some actions. A change to a global constraint forces us to modify the actions.

Clearly, we could modify the action in order to satisfy the global constraints. But, in real situations this is impractical; the specification is commonly written by software engineers or experts who have a deep knowledge of the software which the system administrator usually does not have. Determining whether an action must be modified or not may be as hard as the planning itself, since the constraints for execution could require arbitrary states to be achieved by previous actions. In addition, a modification may not be allowed due to a lack of permission or a license violation, for example.

On the other hand, current declarative configuration languages do not have the notions of actions and global constraints. Planning Domain Definition Language (PDDL) [McDermott et al., 1998], as de facto language of the automated planning community, cannot be used because it lacks of desired features which are available in the practical configuration languages such as object-oriented modelling, inheritance, composition, and file inclusion. A mixed representation is not a good solution either because the system administrator must learn two languages at the same time, where both have dif-

[1] This could be automatically generated by the tool.

[2] The architecture has: a *planner* that use *goals* (from users) and *system descriptions* (objects and actions descriptions) as well as the current *state* of the system (from *sensor*) to automatically generate a *plan* to achieve the goals; a *controller* that deploys configuration changes by executing *actions* within particular ordering constraints as defined in the *plan*, it also observes the state (from sensor) of the system as plan being executed in order to monitor particular uncertainty of the system. Note that some *events* may change the state of the system. Thus, it is necessary for the configuration to perform a continuous deployment to correct any drift from the goals.

[3] In the previous works, we can only define the constraints that must be satisfied at the final state as the goals.

ferent semantics. This is clearly error-prone.

These issues lead us to the first and second hypotheses of this thesis:

> **Hypothesis 1**: "It is possible to extend the existing practical configuration language which could be used to define a configuration specification which consists of the desired state, the actions, and the global constraints."

> **Hypothesis 2**: "It is possible to automatically generate a workflow which can bring the system from the current to the desired state, but also preserve global constraints during configuration changes."

On the other side, most practical configuration tools are highly centralised – a central controller gathers information about the state of the system and orchestrates the workflow by communicating directly with the systems involved at each step. Although this is simple and easy to implement, this creates a potential bottleneck in large systems, and can also be unreliable if the communication with the controller is disrupted, which is particularly relevant since reconfiguration frequently occurs as an autonomic response to system failures.

Fully distributed planning[4], on the other hand, is not a good solution to this problem either – avoiding deadlock/livelock may require agents to have considerable global knowledge and achieving such knowledge is likely to result in even more costly inter-agent communication. Predicting the behaviour of such systems is also more difficult, and hence they are less acceptable to system administrators in real situations.

SmartFrog [Goldsack et al., 2009] is a declarative configuration tool that has an interesting exception in addressing this issue. Its resource components which manage the various aspects of the system can be augmented with *Behavioural Signatures* [Farrell et al., 2010]. These allow the system administrator to specify a set of models which defines state-dependencies between components so that a change of state in one component may depend on changes of state in other components. This could produce a cascade of distributed state changes with particular ordering constraints. Unfortunately, the dependencies must be computed manually which is error-prone and time consuming. The models must also be validated for livelock and deadlock conditions before they can be deployed. And if such condition is found, then they must be corrected manually, which makes it impractical.

These motivate our third hypothesis:

[4]The system is managed by a set of agents, each of which can automatically generate and execute arbitrary workflow.

> **Hypothesis 3**: "It is possible to automatically construct a set of reactive agents augmented with Behavioural Signatures, that choreograph configuration changes without any central controller."

The above hypotheses are related with a conceptual architecture presented in figure 1.1. They concern on different elements of the architecture. Hypothesis 1 concerns on increasing the expressiveness of the configuration language which is used to specify the *goals* and the *descriptions* of the system. Hypothesis 2 focus on the capability of the *planner* for solving a configuration task with global constraints. And hypothesis 3 provides an alternative approach of the *controller* for deploying configuration changes.

1.1 Contributions

Based on the hypotheses, the contributions of this thesis can be divided into three parts: modelling, planning and deployment. The first part is for supporting the first hypothesis by extending an existing configuration language to add the notion of global constraints. Although this may look simple, but this is a non-trivial problem since most popular configuration languages are extremely informal where the behaviour is defined implicitly by a single implementation. If we are not carefully adding a new feature to the language, then it could break the existing semantics or introduce ambiguities. Thus, the first part of our contributions are:

- Formally define the semantics of the core subset of SmartFrog [Goldsack et al., 2009] language.
- Extending the semantics of the core subset of SmartFrog by adding new notations for the action description and the global constraints.

The second part is related to the second hypothesis which is automatically generating a workflow that can achieve the desired state and preserve the global constraints. However, although this thesis is mainly motivated by problems of system configuration domain, but one of our contributions can be applied to any domain. These contributions are:

- A domain independent technique to compile a planning problem with extended goal into a classical planning problem.
- A technique to solve a configuration task by translating the task into a classical planning problem.

The last part of the contributions is related to the third hypothesis which is a mixed approach between the centralised planning and the distributed execution for deploying configuration changes. The contributions are:

- A choreographing technique that automatically constructs a set of reactive agents augmented with Behavioural Signatures.
- A multiagent regression algorithm that executes a plan distributively, generalises the plan to reduce the requirements of replanning, and enables the self-healing capability.

Another contribution is a prototype system configuration tool, called Nuri, that implements the techniques described in this thesis. It is released as an open source software which can be accessed at:

`http://homepages.inf.ed.ac.uk/s0978621/nuri`

Nuri implements a conceptual architecture as illustrated in figure 1.2. The main difference of this architecture comparing to a centralised one (e.g. figure 1.1) is that it assumes the system consisting of *subsystems*, each of which is managed by an *agent*

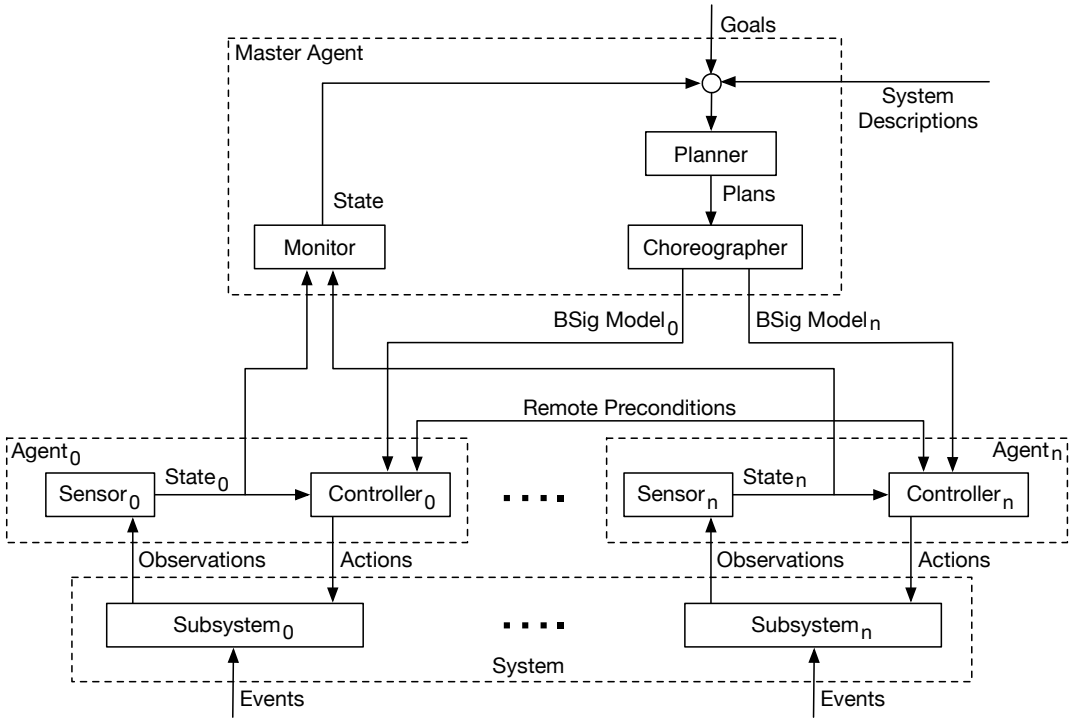


Figure 1.2: A *choreography* architecture of Nuri system configuration tool (BSig model = Behavioural Signature model).

which has a *sensor* and a *controller*. There is a master-agent that has: a *monitor* that aggregates the states of subsystems from all sensors; a *planner* that uses the state (from *monitor*), goals (from users), and system descriptions to automatically generate a plan that can achieve the goals; a *choreographer* that automatically translates the plan into a set of Behavioural Signature (BSig) models, each of which is sent to a particular agent. Each agent's *controller* continuously observes the state of the system (from *sensor*) in order to select and execute appropriate actions in order to achieve the subsystem's goal as defined in the BSig model as well as anticipate any uncertainty on the subsystem.

1.2 Thesis Structure

The structure of this thesis is illustrated in figure 1.3. It consists of seven chapters whose relations are shown by arrows. The first chapter presents the motivations, the hypotheses and the contributions of this thesis. It is then followed by a chapter that summaries the background and the related work on the system configuration and the automated planning.

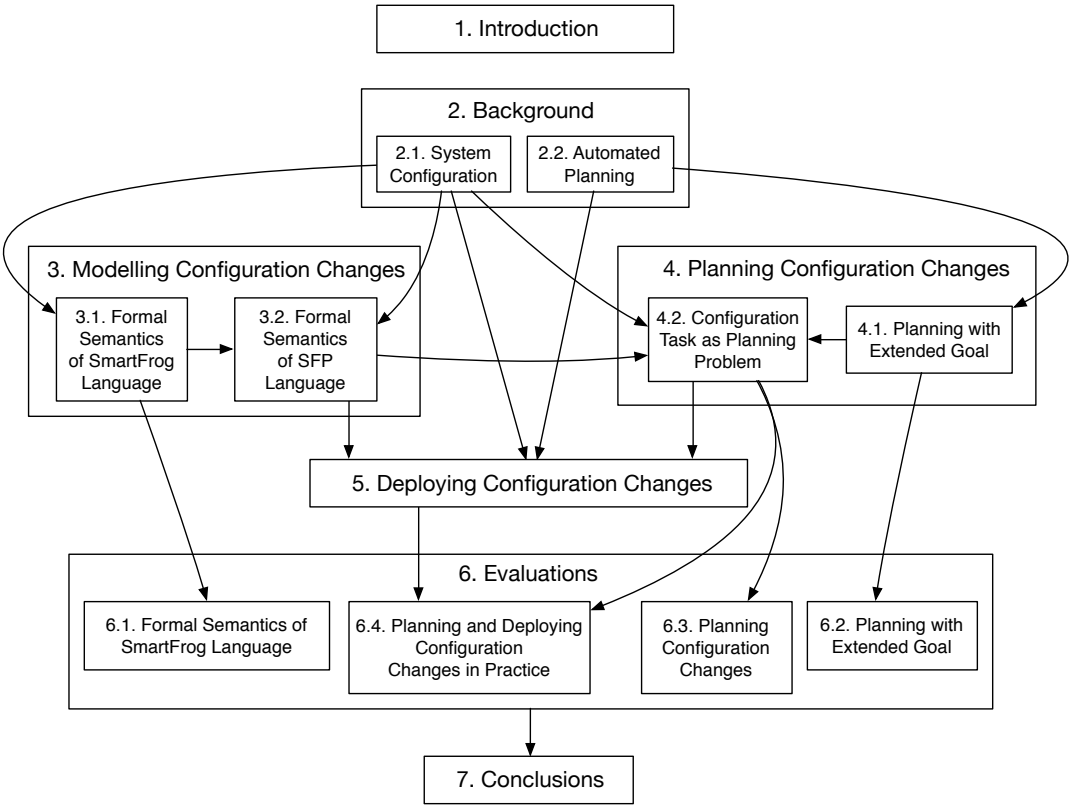


Figure 1.3: The structure of chapters and sections and their relations (arrows).

Contributions on modelling configuration changes are presented in chapter three. It describes the formal semantics of the core of SmartFrog (SF) language. It then presents the semantics of SFP language that extends this SF core. This extension will be used in other chapters.

Chapter four presents the second part of contributions on planning. The first section describes a domain-independent technique to compile a planning problem with extended goal into a classical planning problem. The second section presents a technique to translate a configuration task into a classical planning problem whose solution (plan) will be executed for implementing the configuration changes.

In chapter five, a novel technique for deploying configuration changes is presented. First, it describes how to construct a set of reactive agents that choreograph the plan execution without any central controller. Second, it presents a multi-agent execution algorithm for executing the plan with regression that enables the self-healing capability.

To validate the contributions, the evaluation results are described in chapter six. Finally, we present the conclusion and the future works of this thesis.

1.3 Related Publications

Some contents of this thesis have been published on the following papers:

- H. Herry and P. Anderson, Planning Configuration Relocation on the BonFIRE Infrastructure, In *Proceedings of CloudCom 2013 Workshop on Using and Building Cloud Testbeds (UNICO'13)*, 2013.

- H. Herry, P. Anderson, and M. Rovatsos, Choreographing Configuration Changes, In *Proceedings of 9th International Conference on Network and Service Management (CNSM'13)*, 2013.

- P. Anderson, S. Bijani, and H. Herry, Multi-agent Virtual Machine Management Using the Lightweight Coordination Calculus, In *Transactions on Computational Collective Intelligence XII*, pages 123-142, 2013.

- H. Herry and P. Anderson, Planning with Global Constraints for Computing Infrastructure Reconfiguration, In *Proceedings of the AAAI-12 Workshop on Problem Solving using Classical Planners (CP4PS'12)*, 2012.

- H. Herry, P. Anderson, and G. Wickler, Automated Planning for Configuration Changes, In *Proceedings of 25th Large Installation System Administration Conference (LISA'11)*, 2011.

Chapter 2

Background

This chapter presents the related backgrounds and the literature survey of two research fields which are System Configuration and Automated Planning, as this thesis is on the junction between them.

2.1 System Configuration

System configuration deals with a problem to transform a set of computing resources into a functioning system according to particular requirements. Basically, the problem starts with a set of bare metal machines, a repository of the necessary software packages and data files, and a specification of the functions that the entire system should perform based on the requirements. Then, we must load the software and configure the machines in order to enable the required functionality. Whenever the specification or the environment[1] changes, then we must reconfigure the machines to maintain conformance with the specification [Anderson, 2006].

Although the above basic problem is straightforward, but in practice, there are several factors that complicate the problem and make it very difficult to be solved.

One of the factors is managing the relationships. In system configuration, we are mainly concerned about the functionality of the whole system which involves complex relationships between configurable components, where the configuration of a particular component can depend on the configuration of another component. For example, to have a working web application, we must not only configure a web server, but also a database server, the firewall settings, etc; the web server itself must know the port number that the database server will listen to. Some relationships exist between com-

[1] For example, some application services are broken.

ponents on the same machine, while the others exist between components on different machines. The latter is the most difficult situation because managing relationships in a distributed system is very complex.

Another complicating factor is managing changes. Change management is required because both the configuration specification and the system itself are in a constant state of change. The rate of change can vary between different systems. The changes can be caused by many reasons, for examples: the software packages must be updated to fix some security vulnerabilities; new machines must be configured to replace the broken ones; a new instance of web service must be added to the cluster to address the increasing demands. The relationships between components increase the complexity of change management since any change on a particular component can affect the others – we may not want to stop a server until all clients have redirected to using another one, for example. Thus, it is a good practice to carefully plan and verify every configuration change before deployment to avoid any unexpected outcome.

Uncertainty is also a factor that complicates the system configuration problem. It is natural that there is no certainty in the system's environment e.g. every machine can be broken at any time. On the other hand, there is no guarantee that every configuration change will be successful because the configuration process can fail at any step. Thus, a good configuration tool should be able to reason about this uncertainty to have a sound reconfiguration of the system. It also must be sufficiently robust to any configuration failure.

[Kephart and Chess, 2003] see the problem as a grand-challenge, and introduced an idea of *Autonomic Computing* which can be summarised as follows:

> *"Like their biological namesakes, autonomic systems will maintain and adjust their operation in the face of changing components, workloads, demands, and external conditions and in the face of hardware or software failures, both innocent and malicious."*

The essence of autonomic computing system is *self-management* with the intention to free the administrator from the details of operating and maintenance processes. Any computing system should be able to manage itself given high-level objectives from administrators. There are four aspects of self-management that should be provided by the system:

- *self-configuration* – the system automatically and seamlessly adjusts itself to maintain conformance to the high-level policies;

- *self-optimisation* – the system continually searches opportunities to improve its performance and efficiency;
- *self-healing* – the system automatically detects, diagnoses and repairs any software and hardware problem;
- *self-protection* – the system automatically defends against malicious attacks or cascading failures.

Supporting the autonomic system introduces new complications, in particular on how to specify the desired configuration. If the specification has too many low-level details, then there is not enough room for the configuration tool to find an alternative solution. For example, we might specify:

- Server A and B run an identical web service;
- Machine X uses a web service of server A;
- Machine Y uses a web service of server B.

However, whenever server A fails, then machine X will fail as well because there is no alternative web service in our specification. We may use the following event-condition-action (ECA) rules to define procedures for failure recovery:

- If server A fails, then set machine X to use a web service of server B;
- If server B fails, then set machine Y to use a web service of server A.

Unfortunately, this is not a good solution because there is no explicit desired specification because the rules will dynamically change the specification. A better solution is to define the specification as a logical formula:

- Server A and B run an identical web service;
- Machine X uses a web service of server A or B;
- Machine Y uses a web service of server A or B.

Based on the above specification, the tool can automatically compute the configuration details as well as the required steps that make true the desired configuration.

2.1.1 Approaches

The approaches to address the system configuration problem have evolved from manual to more automated approaches. The following paragraphs briefly highlights this evolution [Herry et al., 2011].

Manual configuration – the administrator manually computes the necessary actions to change from one to another configuration, and then manually executes the commands to implement this. Clearly, this is error-prone and time-consuming. It is difficult to prove reliably that this manually chosen sequence of changes will match with the configuration requirements under all circumstances.

Imperative scripts – this is similar to the previous approach, except that the sequence of changes is captured in an imperative script. This is allowing it to be executed multiple times and on different systems, or to be shared between the system administrators to be reused or modified for particular systems. This clearly makes it easier to deal with large numbers of systems, and until recently, this was probably the most common approach to configuration for many people. However, this approach still suffers most of the problems faced by the manual approach. In particular, it is not always obvious that the selected script could meet the necessary preconditions. Thus, the system administrator has a tendency to apply the script blindly resulting in very unpredictable outcomes.

Fixed workflow orchestration – in the case where the sequence of configuration changes really matter, some tools are able to perform sequences of configuration changes automatically and/or unattended. A system administrator can create or choose a workflow[2] from the workflows repository which is suitable with the current and the desired state of the system. Then, the execution of the workflow will be orchestrated by a central controller by invoking the actions remotely using particular protocol such as *secure remote shell*. This helps the administrator to easily configure the machines in various locations. This is also suitable for a typical organisation that has centralised management. However, this approach still requires the administrator to compute the workflows manually, where a very large number of workflows will be required to cater for every eventuality, even for a small system. In addition, this is conceptually similar to the imperative scripts where choosing an appropriate workflow is not always obvious. Ansible [Ansible Inc., 2014], IBM Tivoli [IBM Corp., 2014], and Microsoft System Center [Microsoft Corp., 2014] are the examples of configuration tools that implement this approach.

[2]We might have to set some necessary parameters as well.

Declarative specification – some popular configuration tools , such as Puppet [Puppet Labs, 2014], SmartFrog [Goldsack et al., 2009], CDDLM [Loughran and Toft, 2008], and LCFG [Anderson and Scobie, 2002], are using a kind of declarative approach which allows us to explicitly specify the desired state. This specification is then used by the tool to compute and implement the necessary changes. The tool guarantees that the final state would match the required specification, regardless of the current state of the system. The changes are implemented distributively by a set of agents, each of which is controlling a particular machine. However, these tools do not maintain the sequence of configuration changes between different machines – the changes are essentially implemented in indeterminate order. To perform such a sequence, the administrator must manually present the intermediate and the final states step-by-step to the configuration tool. This is clearly impractical and cannot be used in unattended situation.

Dynamic workflow orchestration – some previous works [El Maghraoui et al., 2006, Levanti and Ranganathan, 2009, Hagen et al., 2009, Hagen and Kemper, 2010, Di Cosmo et al., 2012, Lascu, 2014] have used automated planning technology to bridge the gap between a declarative specification and the fixed workflow orchestration. This approach allows us to model the desired configuration state of the system as well as the operational capabilities (actions) of the resource components. Then, the model is given to the planner which will automatically generate the workflow, eliminating the need of an administrator to compute it manually. A central controller will orchestrate the workflow by scheduling what action should be executed when. However, none of the state of the art techniques allows us to define the global constraints – the constraints that should be satisfied in the intermediate and the final states. In addition, the centralised deployment architecture suffers several issues such as bottleneck and single point of failure.

Event-driven workflow – some practical configuration tools, such as Juju [Canonical Ltd., 2014], implements an event-driven workflow approach, which is based on the *reactive programming* paradigm[3]. In this approach, every resource component has lifecycle states. Whenever the component is making a transition between two states, it will raise an event that will be caught by dependent components that can perform particular actions as a reaction to the state change. This will produce a cascade of changes between the resource components. Unfortunately, the event-model must be validated for

livelock and deadlock conditions before deployment. And if such condition is found, then it must be corrected manually which is not an easy task.

The above comparison of approaches is summarised in table 2.1.

2.1.2 Configuration Language

Regardless of the approach that we choose, the specification plays a vital role in the system configuration because it drives the administrator and/or the tool in making the configuration changes in order to provide the required functionality. In practice, the specification only determines aspects that we care about, which may vary between machines or may change over time. The specification could be simply defined in any human language. However, if we use a configuration tool, then the requirements must be translated into a specification in a particular configuration language supported by the tool.

As described above, there are a broad range of approaches to the tools and their languages. At one side, the language may be an extension of an imperative scripting language. For example, Chef [Opscode Inc., 2014] is using a domain-specific language based on Ruby. At the other side, it may be a custom domain-specific language which allows a declarative specification of the desired state, such as Puppet [Puppet Labs, 2014] and SmartFrog [Goldsack et al., 2009] languages.

Imperative languages are easier to be adopted by system administrators since it is naturally an extension of an existing manual or scripted process. They usually have little or even no explicit specification of the desired configuration state. The administrator uses the language to manually define workflows that should be executed during the deployment process to achieve the desired configuration. Although it is easier to control the sequence of configuration changes, the workflows need to be tested carefully to make sure that they can achieve the (implicit) desired state. The workflow execution results could be brittle if the initial state of the system is one that has not been anticipated.

On the other hand, a *custom* declarative language allows us to explicitly define a specification of the desired state which is independent of the deployment process – for example, Puppet [Puppet Labs, 2014]. Unlike the imperative language, this makes us easier to specify and reason about the desired configuration, although it is harder

[3]Reactive programming is a programming paradigm which is built on the propagation of change where state changes are automatically propagated across the network of dependent computations by the underlying execution model[Bainomugisha et al., 2013].

Practical Tool / Work	Specification			Ordering Constraints	Deployment		
	Paradigm	Language	Global Constraints		Architecture	Method	Continuous
Ansible	Imperative	YAML DSL	No	Manual	Centralised	Push	Yes
IBM Tivoli	Imperative	WorkFlow	No	Manual	Centralised	Push	Yes
Ms. System Center	Imperative	Runbooks	No	Manual	Centralised	Push	Yes
Puppet	Declarative	Puppet	No	Manual*	Weakly distributed	Pull	Yes
SmartFrog	Declarative	SmartFrog	No	Manual**	Strongly distributed	Push	Yes
CDDLM	Declarative	CDL	No	Manual***	Strongly distributed	Push	Yes
LCFG	Declarative	LCFG	No	Not available	Weakly distributed	Pull	Yes
Juju	Declarative + Event-model	YAML + Shell-Script	No	Manual	Weakly distributed	Push	Yes
[El Maghraoui et al., 2006]	Declarative + Action-model	Codasyl DDL + PDDL	No	Automated	Centralised	Push	Yes
[Levanti and Ranganathan, 2009]	Declarative + Action-model	Tags + SPPL	No	Automated	Centralised	Push	No
[Hagen and Kemper, 2010]	Declarative + Action-model	Groovy DSL	No	Automated	Centralised	Push	No
[Di Cosmo et al., 2012]	Declarative + Action-model	JSON DSL	No	Automated	Centralised	Push	No
[Lascu, 2014]	Declarative + Action-model	JSON DSL	No	Automated	Centralised	Push	No
Nuri (this thesis)	**Declarative + Action-model**	SFP	**Yes**	**Automated**	**Weakly distributed**	**Push**	**Yes**

Figure 2.1: A comparison summary of approaches to the system configuration problems. Abbreviations: DSL = Domain Specific Language; CDL = CDDLM Definition Language; DDL = Domain Definition Language; PDDL = Planning Domain Definition Language; SPPL = Stream Processing Planning Language; YAML = Yet Another Markup Language; JSON = JavaScript Object Notations. $(^{*})$ The ordering constraints between resource components within a single machine, but not across different machines, can be defined using Puppet *relationship*. $(^{**})$ The ordering constraints can be defined as the Behavioural Signature model. $(^{***})$ The ordering constraints between lifecycles of CDDLM components can be defined using CDL control-flow.

to control the sequence of configuration changes. However, automated planning techniques now can be used to generate workflows that are guaranteed to satisfy any required constraints, and achieve the desired state from any viable initial state [Herry et al., 2011].

For pragmatic reasons, some configuration languages are closely related with the underlying implementation and the deployment. For example, Chef [Opscode Inc., 2014] specification, which is defined in Chef configuration language, can contain arbitrary Ruby code. The semantics of this kind of languages is likely to require a complete semantics of the embedded language, and unlikely capture the higher level meaning of the specification. For the case of Chef language, since arbitrary Ruby code can be embedded to the specification, then its semantics must also include a complete semantics of Ruby language.

The advantage of a separate language is to allow us to focus on clean statements of the configuration requirements. It could provide specific features for effectively supporting the tasks of system administrators such as sharing reusable configuration elements, composing various elements of configuration from different resources, and providing a *loose*[4] specification [Hewson et al., 2012]. The configuration specification can be deployed using a separate independent language which is the most appropriate language of the tool. This clear separation should make the configuration language to have a simpler semantics, and most importantly able to capture the essence of the requirements. In addition, the semantics of the language can guarantee some desired properties that are not provided by multipurpose programming language e.g. the evaluation of the specification is always terminated[5].

2.1.3 Specification Deployment

After the requirements have been defined in the configuration specification, then a configuration tool is needed to deploy the specification onto the machines. The followings describe aspects related to the deployment process.

[5]In loose specification, instead of defining one specific configuration, the system administrator defines constraints that should not be violated by the configuration. The tool will then automatically select a single configuration, from a set of possible solutions, which satisfies the constraints as well as maximises particular objective functions.

One-time and Continuous Deployment

Some tools only have an ability to deploy the specification from scratch which is called a *one-time* deployment. They cannot be used to reconfigure an existing system, in particular when the specification has changed. For example, RedHat Kickstart [RedHat Inc., 2014] can install the operating system and the required software packages onto target machines, but it cannot be used to add or remove the packages when the machines are running.

On the other hand, some tools, such as Puppet [Puppet Labs, 2014], have an ability to perform a *continuous* deployment. These tools can deploy the specification regardless of the state of the system. When there is a change on the specification, the tools will reconfigure the existing system to implement the change. When there is a change in the system environment, the tools will correct the system to maintain conformance with the specification (*self-healing*). This capability is commonly available in the state of the art of configuration tools.

Idempotent Action

An action is called *idempotent* if executing it multiple times will have the same effect as executing them once [Anderson, 2006]. Whenever the scripts contain some actions that do have this property, then it is necessary to keep track of every change that has been applied to the machines, which is extremely difficult.

Some configuration tools that use a more imperative approach are relying on idempotent actions to enable continuous deployment capability – for example, Chef [Opscode Inc., 2014]. The script with idempotent actions is just simply re-run again to repair any machine whose configurations is not correct.

Deployment Architecture

[Delaet and Joosen, 2010] classified the deployment architectures into three categories. First is a *centralised* architecture where a single agent is running on a central server and it is responsible to deploy the specification onto all machines. This architecture is simple and easy to be implemented. However, when dealing with a large scale system, this central server quickly becomes a bottleneck. It is also less reliable because the system could be out of control whenever there is a failure on the central server.

Other tools are using a *weakly distributed* architecture where every machine has an agent that is responsible to deploy the particular specification onto the machine.

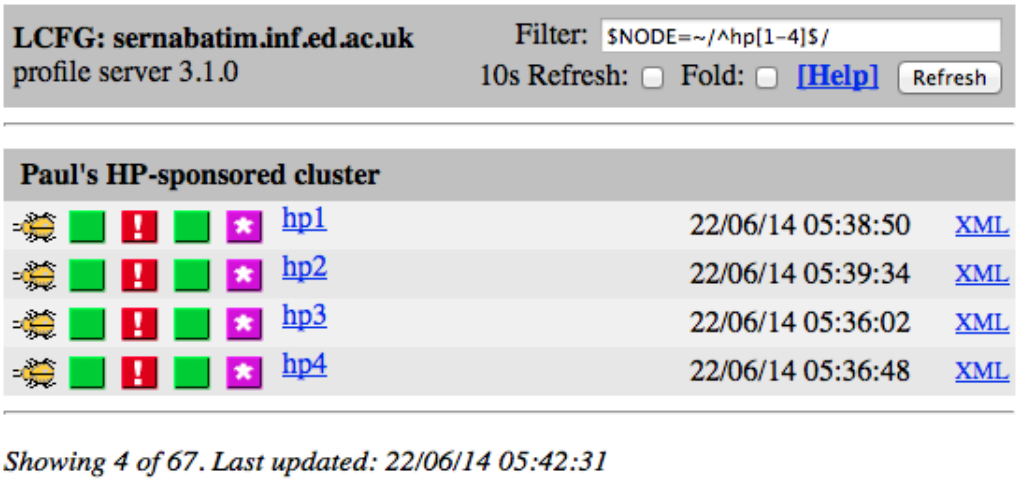


Figure 2.2: The LCFG status display of four machines.

A central server still exists, but it only has functions to compile and distribute the specification to the agents either using a *pull* mechanism – the agent is responsible to pull the specification from the server periodically, or a *push* mechanism – the server periodically pushes the specification to the agents. To address the bottleneck issue, it is possible to replicate the server and share the specification between the servers.

Some tools are implementing a *strongly distributed* architecture where every machine has an agent that has the same capabilities, such as compiling and deploying the specification. Commonly, the system administrator is responsible to push the specification to the agents. The difficulty of this architecture is the coordination between the agents during the deployment process, in particular when there are some ordering constraints that should be maintained across different machines (agents).

2.1.4 Monitoring

In practice, there could be discrepancy between the current configuration state of the system and the desired configuration state defined in the specification at any one time. This is normal as the implication of latency during the deployment process.

A monitoring tool could be used to provide feedback to the system administrator who can manually monitor the deployment process. It helps the administrator to determine whether the specification has been completely deployed, or there is an error during deployment process that needs to be addressed – for example, the administrator can present an alternative specification to address particular failure. Figure 2.2 shows a screen capture of the LCFG monitoring tool that displays the status of four machines.

The information from the monitoring tool can be exploited by an *intelligent* configuration tool to provide a better solution. Instead of manually presenting an alternative specification, the administrator can define a *loose* specification. Thus, the tool can automatically compute the alternative desired state as well as the new workflow in order to address particular failure – for example, a new web service must be installed and started on a new virtual machine since the old one has failed.

2.1.5 Practical Configuration Tools

This subsection presents a survey of some practical configuration tools which are known to be used in production systems. The first three tools, i.e. Ansible, Chef, and Puppet, are very popular – the following table shows the number of Github users that have starred or forked their main repository. On the other hand, SmartFrog is mainly used by Hewlett-Packard for their systems.

August 15th, 2014

Tool	Total Stars	Total Forks
Ansible	7,221	2,221
Chef	2,929	1,200
Puppet	2,561	1,103

2.1.5.1 Ansible

Ansible [Ansible Inc., 2014] was first released in 2012 and quickly gained attention from the users because of its simplicity. It is using a centralised deployment architecture where, unlike other architectures, it does not require a specific agent to be installed and running on the target machines. All parts of the tool are running on a single machine that acts as the central controller. The execution of the actions on the target machines are performed using a *remote shell execution* protocol. However, this architecture comes with a price that there is a bottleneck issue when the tool is used to manage a large scale system.

The configuration specification is defined in a custom imperative language based on YAML [yam, 2014]. We must translate the requirements into an ordered list of actions (workflow) which will be executed by the tool to achieve the desired configuration. The sequence of configuration changes can be controlled simply by reordering the declaration of the actions. Conceptually, this is similar to the imperative scripts where we must compute the workflow manually.

2.1.5.2 Chef

First released in 2009, Chef [Opscode Inc., 2014] is a configuration tool implemented in Ruby. A Chef specification is defined using a domain-specification language based on Ruby. Every requirement must be translated into a specification by defining a set of idempotent actions that should be executed to achieve the desired configuration. By default, the order of configuration changes is determined from the declaration order of the actions. But, this can be overridden by defining partial ordering constraints (relationships) between the actions.

Chef is implementing a *weakly distributed* architecture for deploying the specification where a chef-server acts as the central controller and every target machine is controlled by a chef-client. We can use a *push* mechanism where the chef-server orchestrates the execution of the actions across different machines. We can also use a *pull* mechanism where every chef-client periodically pulls the specification from the server and then deploys it to its machine, but there is no guarantee that ordering constraints are maintained across the machines.

2.1.5.3 Puppet

Initially developed in 2005, Puppet [Puppet Labs, 2014] is perhaps the most popular configuration tool supported by a large open source community. It has a custom declarative configuration language that allows us to explicitly define the desired configuration state of the system. The specification contains instances of *abstract resource* whose attributes are representing the desired state of the resource. Every value of the attributes is set based on the requirements.

Puppet deploys the configuration changes in essentially indeterminate order. But since 2012, the language provides the notation of dependencies between resource components (relationships) which can be used to define a partial ordering constraint of the deployment process between two resource components. In order to avoid a livelock or deadlock situation, the tool will produce an error whenever it detects cyclical dependencies. However, the ordering constraints are only maintained between components on the same machine, but not between different machines. On the other hand, managing dependencies is complex because they are not invariant to the desired state – any change of the desired state may require a change in dependencies.

Puppet is implementing a *weakly distributed* deployment architecture with a *pull* mechanism. To reduce the bottleneck issue, a system can have one or more central

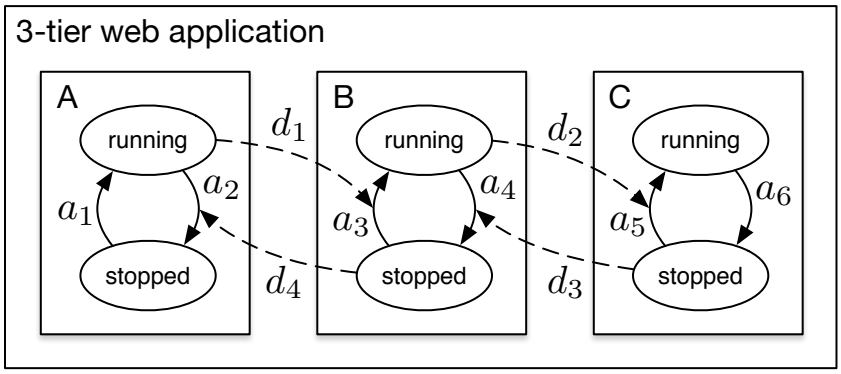


Figure 2.3: An example of Behavioural Signature model for the 3-tier web application.

servers (puppet-master) that serve a large number of agents (puppet-client). Every agent periodically pulls and deploys the compiled version of the specification, called a *catalog*, onto the target machine.

2.1.5.4 SmartFrog

Smart Framework for Object Groups (SmartFrog) [Goldsack et al., 2009] is a distributed configuration tool for configuring and managing distributed software systems. It consists of: a custom declarative configuration language for defining configuration specifications; a daemon agent, which is a secure, distributed (peer-to-peer) controller for managing a particular machine; and components, which are software libraries to implement the configurations. SmartFrog implements a *strongly distributed* deployment architecture. The configuration is deployed by *pushing* the specification to target machines.

Another feature of SmartFrog is the specification of a model called the *Behavioural Signature* [Farrell, 2008] which defines state-dependencies between components so that a change of state in one component may depend on changes of state in other components. This can produce a cascade of distributed state changes with a particular ordering constraint.

Figure 2.3 shows an example of a system that is orchestrated by SmartFrog Behavioural Signature model. The system has three components: A (database layer), B (logic layer), and C (presentation layer), each of which has a set of states and actions. The state-dependencies denoted by d_1, d_2, d_3, and d_4 guard the execution of the actions that change the state of the components. For example, d_1 means that action a_3 can be executed if A is at state *running*. If all components are at state *stopped* and the user sets

that the desired state of C is *running*, then the SmartFrog will automatically execute a_1 and then a_3 to change the state of A and B to *running* before executing action a_5. The state-dependencies are manually defined in the configuration specification. They constrain the *behaviour* of components which determine *what* a component may perform and *when*.

The following is a complete SmartFrog specification for an example in figure 2.3:

```
#include "org/smartfrog/components.sf"
#include "org/smartfrog/services/dependencies/components.sf"
// Prototype component
SimpleComponent extends State {
    sfClass "org.webapps.SimpleComponent";
    // attributes
    name TBD;  // name of component
    run false; // state of component
    // state-transitions
    tStart extends Transition {  // starting the component
        guard (! LAZY run);      // precondition
        effects extends PolicyEffects {
            run true;            // effect
        }
    }
    tStop extends Transition {   // stopping the component
        guard ( LAZY run );      // precondition
        effects extends PolicyEffects {
            run false;           // effect
        }
    }
}
// Prototype of dependencies between two components
startDependency extends Dependency {
    enabled ( LAZY on:run );   // effect
    relevant ( LAZY by:run );  // condition
}
stopDependency extends Dependency {
    enabled (! LAZY on:run );  // effect
    relevant (! LAZY by:run ); // condition
}
// Main component contains the specification for WebApps
sfConfig extends Model {
    // artifacts
    A extends SimpleComponent {
        name "A";
    }
    B extends SimpleComponent {
        name "B";
    }
    C extends SimpleComponent {
        name "C";
    }
    // dependencies between artifacts
    BStart extends startDependency {
        on LAZY B;  // implies: enabled (LAZY B:run)
```

```
47          by LAZY A;  // implies: relevant (LAZY A:run)
48      }
49      CStart extends startDependency {
50          on LAZY C;  // implies: enabled (LAZY C:run)
51          by LAZY B;  // implies: relevant (LAZY B:run)
52      }
53      AStop extends stopDependency {
54          on LAZY A;  // implies: enabled (LAZY A:run)
55          by LAZY B;  // implies: relevant (LAZY B:run)
56      }
57      BStop extends stopDependency {
58          on LAZY B;  // implies: enabled (LAZY B:run)
59          by LAZY C;  // implies: relevant (LAZY C:run)
60      }
61  }
```

The above specification includes (lines 1-2) two files. Lines 4-22 defines prototype `SimpleComponent` which is a common component that will be inherited by other components. It has attribute `sfClass` that specifies a Java classpath which contains the component implementation that will be instantiated by SmartFrog runtime system. It also has attribute `name` (the component's name) and `run` (`true` if the component is running, otherwise `false`). Their default values are `TBD` (To Be Defined) and `false` respectively. In addition, the prototype has two state-transitions: `tStart` (lines 10-15) changes the component's state from "stopped" (line 11) to "running" (line 13), and `tStop` (lines 16-21) changes the component's state from "running" (line 17) to "stopped" (line 19).

The specification has two other prototypes (lines 23-31) that defines common dependencies between two components. These prototypes has two attributes whose relationship is that the evaluation value of `relevant` must be true before the evaluation value of `enabled` is true. Lines 24-27 define *stop* dependency: the component referred by `relevant` must be stopped before stopping the component referred by `enabled`. Lines 28-31 define *start* dependency: the component referred by `relevant` must be running before starting the component referred by `enabled`.

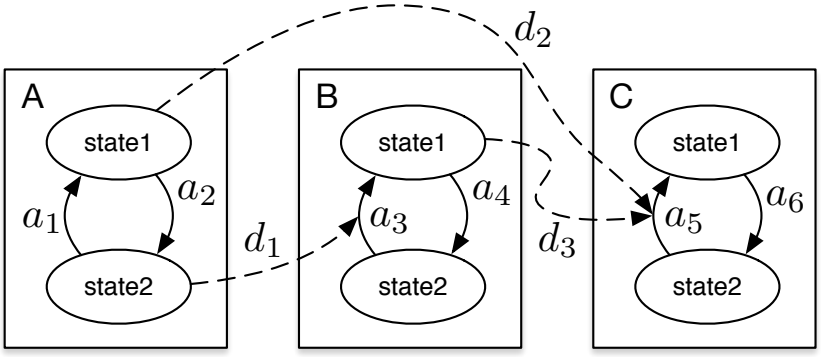
The last part (lines 33-61) define the main component `sfConfig` which contains the specification of the system. Lines 35-43 define component A, B and C. Every component uses `SimpleComponent` as prototype, inheriting attribute `sfClass`, `name`[6] and `run`, and also state-transition[7] `tStart` and `tStop`. Lines 45-48 defines dependency d_1, lines 49-52 defines dependency d_2, lines 53-56 defines dependency d_4, and lines 57-60 defines dependency d_3.

[6] The value of `name` of each component has been overridden to `"A"`, `"B"` and `"C"`.

[7] Based on figure 2.3, `A:tStart` is a_1, `A:tStop` is a_2, `B:tStart` is a_3, `B:tStop` is a_4, `C:tStart` is

The Behavioural Signature model enables SmartFrog to orchestrate the state-transitions of components in a distributed way. Unfortunately, the user must compose the model manually by explicitly defining the state-dependencies in the configuration specifications. This is clearly error-prone. Although we can use a model checker to ensure that there is no deadlock/livelock situation, but the resolution of such problem must be done manually, which is impractical.

On the other hand, during deployment, every SmartFrog component is controlled by a thread which uses a backward-chaining mechanism to achieve the conditions of dependency before applying a state-transition. The communications between threads of different components are performed using Java Remote Method Invocation (RMI). For a system that has simple dependencies (e.g. the above system), the implementation of configuration changes is straightforward. However, if a particular component of the system has a state-transition which has two or more dependencies, then there is no guarantee that the system transitions are always performed in correct orders. This issue could bring the system to be unable achieving the desired state.



For example, assume we have a system as illustrated in the above figure. The current state of the system is:

`A.state2`, `B.state2`, `C.state2`

And the goal state is:

`A.state1`, `B.state1`, `C.state1`

Using a backward-chaining mechanism, C sends messages to A and B concurrently in order to attain dependency d_2 and d_3 before executing a_5 for achieving the goal. Hence, there are two possible execution sequences:

1) $a_3 \rightarrow a_1 \rightarrow a_5$

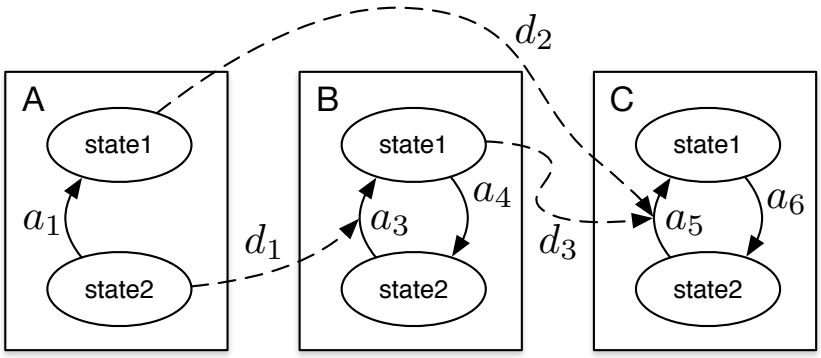
2) $a_1 \rightarrow a_2 \rightarrow a_3 \rightarrow a_1 \rightarrow a_5$

---

a_5 and `C:tStop` is a_6.

Note that the second sequence is possible if A responds to C's request first (d_2) before receiving another request from B (d_1[8]). Both sequences are correct since they can achieve the goal and maintain the dependencies. However, the second sequence may not be desirable if the execution time of a_1 is significant.

On another case, the second sequence is incorrect if a particular action involved in the transitions is *irreversible* (e.g. deleting a file). The following figure provides an example system where this situation may arise.



If A reacts to C's request first, then the system transitions will be: $a_1 \rightarrow$ **dead-state**. Solving this problem requires the user to manually define an explicit ordering constraint between dependencies, in this case: $d_3 \prec d_2$. Unfortunately, this is non-trivial because if there are n dependencies then the number of possible orders is $n!$. The problem can be more complex if the system has cascading dependencies. An alternative solution would be adding an extra dependency where `B.state1` is the condition of action a_1. Unfortunately, determining which new dependency that has to be added is not always obvious whenever it is done manually, in particular for large scale system. In addition, if we are not careful, adding a new dependency can raise a deadlock/livelock condition to the system.

HP Cells

One of the systems which uses SmartFrog is HP Cells. It is a system that provides a virtual infrastructure (virtual machines, virtual networks and virtual storages) which is managed entirely using declarative configuration approach [Hewlett-Packard, 2008]. The key feature of Cells is that users can model a desired configuration state of their system, push the model to Cells, and then Cells will automatically deploy and maintain the system. Hence, if the user wants to change the configuration of the system, it can

[8]Dependency d_1 means that A should hold its current state (`A.state2`) until B has finished executing transition a_3

be done simply by changing the model. In addition, Cells can automatically fix any drift from the desired state. For usability, Cells provides HTTP RESTful APIs through Cells as a Service (CaaS).

Cells also focuses on security and privacy [Frederic Gittler, 2012] by creating an isolated virtual infrastructure for each user called as *cell*. For example, every network connection from internet to virtual machines of a particular cell, or between virtual machines of different cells, must be explicitly defined in the configuration model. If such connection does not exist in the model, then no connection is allowed.

The heart of Cells is the System Cell which runs across all physical machines, each of which must be running a hypervisor [Banerjee et al., 2012]. Every System Cell contains two types of component: the *host-manager* is a component that manages every action on a physical machine; the *core-system* is a component that manages a particular resource such as network or storage. These types of components are implemented as SmartFrog components which run on SmartFrog runtime systems. Whenever CaaS receives a model of desired configuration state from a user, it passes the model to particular host-manager and core-system, which will perform some actions to deploy the model.

2.1.5.5 Configuration Description, Deployment, and Lifecycle Management

Global Grid Forum (GGF) has been developing specifications of Configuration Description, Deployment, and Lifecycle Management (CDDLM) [Loughran and Toft, 2008]. It is a set of specifications intended to become standards to deploy and manage the lifecycle of services on the Grid computing infrastructure. These standards are platform independent. They have been implemented by several organisations such as HP Labs (Java[9]), UFCG[10] (Java), NEC (Java), and Softricity[11] (.NET).

CDDLM standards has three specifications:

- a Configuration Description Language (CDL) specification, which is an XML based language to declaratively describe the configuration state of a system as a hierarchy of components;
- a Deployment API, which is a set of interface that receives a CDL specification, and then deploys and manages the lifecycle of a system described in the CDL specification;

[9] This implementation is based on SmartFrog.
[10] Unversidade Federal de Campina Grande, Brazil.
[11] It is now part of Microsoft.

- a Component Model, which defines interfaces of software components that must be provided in order to be managed by CDDLM deployment system.

In their assessment report, [Dantas et al., 2006] mentions that CDDLM helps them to provides a standardised way for deploying and managing the lifecycle of systems on different infrastructures. Using an EPR (EndPoint Reference)[12], any properties of component can be easily queried and changed when it is necessary. They can also explicitly specify the relations between components and reuse the component definitions using CDL prototyping mechanism. However, as CDL is an XML-based language with rich and complex features, they have difficulties on reading and editing a CDL specification due to lack of practical tool to help them identifying configuration errors, which increases the learning curve of CDL.

It is possible to enforce particular ordering constraints of lifecycle transitions between CDDLM components of a system. These constraints must be manually defined as a workflow in the CDL specification using *control-flow* notations such as `sequence` and `flow` [Schaefer, 2006]. Clearly, this is conceptually similar to the imperative scripts.

2.2 Automated Planning

Automated Planning[13] is an area of artificial intelligent (AI) which studies the automatic deliberation process that chooses and organizes actions by anticipating their expected outcomes to achieve as best as possible some presented objectives [Ghallab et al., 2004]. There are various forms of planning based on the type of the actions, for examples:

- Path and motion planning – this type focuses on synthesizing of a geometric path in particular space from a current to a destination place which maintains some trajectory constraints along that path that specifies the configuration space of an entity such as a truck, a mechanical arm, or a robot;

- Perception planning – this type concerns on generating plans for gathering information using sensing actions;

[12]The EndPoint Reference is an interface to manage a deployed CDDLM component [Schaefer, 2006].

[13]It is also known as “AI Planning”.

- Navigation planning – it combines the motion and perception planning in order to reach a goal or to explore an area. Mars rovers exploration is an example of projects that uses this type of planning;

- Manipulation planning – the main objective of this type is handling objects for building or reconfiguring them e.g. to assembly a rocket space or a car production line;

- Communication planning – this type deals with dialogs and cooperations between several agents or humans in order to achieve particular objectives e.g. cars sharing system.

A natural approach for solving planning problems are to address the problems with specific representations and techniques. These *domain-specific* approaches are well justified and successful in most of the above application areas. However, they are not addressing some commonalities of all these forms which are required to understand the essence of the planning process itself which can help improve the domain specific approach. In practice, it is easier to adapt some general tools to a specific problem instead of addressing every planning problem anew.

For these reasons, automated planning concerns on *domain-independent* approaches to solve planning problems. Formally, a planning problem can be defined as $\mathcal{P} = (\Sigma, s_0, S_g)$, where $\Sigma = (S, A, \gamma)$ is a state transition system, S is a set of all possible states, $s_0 \in S$ is the initial state, and $S_g \subset S$ is a set of goal states, A is a set of actions, γ is a state transition function, find a sequence of action $\pi = \langle a_1, a_2, ..., a_k \rangle$ corresponding to a sequence of state transitions $\langle s_0, s_1, ..., s_k \rangle$ such that $s_1 = \gamma(s_0, a_1), s_2 = \gamma(s_1, a_2), ..., s_k = \gamma(s_{k-1}, a_k)$, and $s_k \in S_g$ [Ghallab et al., 2004].

A *classical planning problem* uses a *restricted* planning model based on the following assumptions [Ghallab et al., 2004]:

A0. Σ has a finite set of states.

A1. Σ is *fully observable*.

A2. Σ is *deterministic*.

A3. Σ is static.

A4. The planner handles only restricted goals which are specified as an explicit goal state s_g or a set of goal states S_g.

A5. A solution plan to a planning problem is a total-order finite sequence of actions.

A6. All actions and events are instantaneous.

A7. The planner ignores any change that may occur in Σ during planning time.

With the above restrictions, a planning problem may look trivial – the problem is solved simply by searching for a path in a graph. However, even for a simple problem, the size of the graph Σ can be very large. [Bylander, 1994] shows that in general, finding a solution plan of a planning problem is a PSPACE-complete problem.

On the other hand, it is hard if not impossible to use the above assumptions to solve *real-world* planning problems. For example, planning problems for reconfiguring systems of Infrastructure as a Service[14] are *fully-observable*, *non-deterministic* (the system can be at more than one possible current states) and *dynamic* (the system can be reconfigured to a variable size [Vaquero et al., 2008]).

One of approaches to the *non-deterministic* problem has been using an execution monitoring that observes the world state as a plan being executed, and then uses this to help the controller selecting appropriate actions of the plan [Muise et al., 2011, Shah et al., 2007]. This can increase the *viability* of the plans under particular uncertainty and reduce the needs of re-planning. §5.2 describes an alternative technique that uses this approach.

An approach to the dynamic size problem is allowing the user to define a *loose* specification. Based on the current state generated by the sensor, the tool can automatically computes a particular desired state by scaling up/down the system in order to achieve particular objectives. [Hewson et al., 2012] is an example work that has been using a constraint solver to generate such desired state. In order to have a complete solution, an automated planner can then be used to generate a workflow to bring the system from current to the desired state.

On the other side, it is possible to relax some restrictions of the classical planning model for some reasons, for example: to increase the expressivity of the representation so that the problem objectives can be expressed more natural. §4 presents techniques that relax two assumptions:

- Relaxing assumption A4 (restricted goals) – we might want to specify the *extended goal* of the planning problem where the objectives of the plan not only concern on the final state but also on every visited states;

[14] Infrastructure as a Service (IaaS) is one of types of cloud systems.

- Relaxing assumption A5 (sequential plans) – in some situations, plans whose actions are partially ordered are more desirable than totally ordered e.g. to enable parallelism of actions execution which can reduce execution time.

The approaches of solving planning problem can be classified into two groups. The first is an *optimal planning* which aims to find a plan with the lowest cost that can achieve the goal. And the second is a *satisficing planning* which aims to find any plan that can achieve the goal, regardless of its cost. The optimal planning is preferred whenever the cost of the plan is important. On the other hand, the satisficing planning is preferred whenever the planning time is more important than the cost of the plan – we need to find a plan as quickly as possible.

2.2.1 STRIPS Representation of Classical Planning

In 1971, Richard Fikes and Nils Nilsson developed an automated planner called Stanford Research Institute Problem Solver (STRIPS) to solve planning problems of robotic domain. Later on, the language used in the STRIPS planner became the standard formalism to define a classical planning problem.

In STRIPS, we define a classical planning problem in a first-order language $\mathcal{L}$ which consists of a finite set of predicates $\mathcal{L}_p$, an infinite set of variables $\mathcal{L}_v$, and a finite set of constants $\mathcal{L}_c$. A constant represents an object in the domain. A variable represents any arbitrary constant of the domain. And a predicate represents the relation between objects in the domain.

An atomic formula in $\mathcal{L}$ is a single predicate with constants and/or variables. A ground atomic formula (ground atom) is a single predicate with constants only. A *state* in $\mathcal{L}$ is represented by a set of ground atoms.

A STRIPS planning problem can be formally defined as follows:

Definition 2.1 (**STRIPS Task** [Fikes and Nilsson, 1971])**.**
A planning problem in STRIPS representation (STRIPS task) is given by 4-tuple $\Upsilon = \langle P, A, I, G \rangle$, where:

- P is a finite set of atomic formulas of $\mathcal{L}$;
- A is a set of actions, each action $a \in A$ is a 3-tuple $\langle name, pre, eff \rangle$ where *name* is the name of action, *pre* is a finite set of ground atoms called as the preconditions, and *eff* is a finite set of ground atoms called as the effects;

- I is a finite set of ground atoms called as the initial state;
- G is a finite set of ground atoms called as the goal.

■

There are two types of action preconditions: positive preconditions (pre^+) and negative preconditions (pre^-). If s is the state of the world, then action a is applicable in s if all atoms in positive preconditions are in s, and all atoms in negative preconditions are not in s.

On the other hand, there are also two types of action effects: positive effects (eff^+) and negative effects (eff^-). If action a is applied to state s, then the new state $s' = (s \backslash eff^-) \cup eff^+$ (add all positive effects to, and removes all negative effects from the state). Atoms not mentioned in the effects are assumed not to change during the application of the action (closed world assumption).

2.2.2 Planning Domain Definition Language

Although we can define a classical planning problem using the above formalism, we still need a language to define a planning problem in a more efficient and effective way. In 1998, [McDermott et al., 1998] introduced the Planning Domain Definition Language (PDDL) that was originally designed to be used in the International Planning Competition (IPC). Nowadays, it has become a standard language of the automated planning community to define planning problems from various domains.

PDDL was developed based on STRIPS representation with some additional features that ease us to define a planning problem. Unlike STRIPS, PDDL allows us to define types to limit the possible constants that can substitute a particular variable. For example, the objects `service_a` and `service_b` can all be grouped by the type `service` so that the variable `?s - service` can only be instantiated as one of the two services. Although this does not introduce a new capability[15], but this improves the readability and writing of planning problems.

Another feature is that we can use either *existential* or *universal* quantification in the PDDL action preconditions and goal formula. For example, we can have `(forall (?s - service) (running ?s))` as the goal formula of the planning problem. If the planning domain has two constants of type `service` e.g. `a` and `b`, then the goal formula can be grounded as `(and (running a) (running b))`.

[15] Any typed domain can be rewritten into untyped domain simply by adding a unary predicate for each type along with necessary atoms for constants that uses the type.

The most recent version of PDDL (version 3) [Gerevini et al., 2009] allows us to define a planning problem with *extended goals*. It introduces new notations to express the constraints about the structure of the desired plan (*state-trajectory constraints*) as well as the goal that should be achieved by the plan – this relaxes assumption A4 of the restricted model of the classical planning problem. More details about PDDL3 will be discussed in §4.

In PDDL, every planning problem is defined into two separated files. The first file is describing the domain of the planning problem (PDDL Domain file), which mainly contains lists of types, predicates, and actions. The second file is describing the planning problem itself (PDDL Problem file) which contains the objects, initial state, and the goal state – in PDDL3, the state-trajectory constraints is defined in the PDDL Problem file. Examples of PDDL Domain and Problem files can be found in appendix §C.

In practice, to solve a PDDL planning problem, some planners will first parse the PDDL Domain and Problem files and then transform the problem into a STRIPS-like representation. This transformation will ground the actions by substituting every variable with a particular object based on the type. The planner may also eliminate all first-order (*existensial* and *universal*) quantifications from the actions' preconditions and the goal formulas. The planners then apply particular search techniques to find a solution plan.

2.2.3 Finite Domain Representation of Classical Planning

As mentioned above, some planners are transforming the planning problem into a STRIPS-like representation before starting the search process. This basic representation has helped researchers in developing various successful search techniques. However, some modern planners are using another basic representation which has some properties that can be exploited further for more efficient search techniques. The alternative representation that is commonly used right now is the Finite Domain Representation[16] (FDR) [Helmert, 2009].

FDR is based on the SAS$^+$ planning representation [Bäckström and Nebel, 1995]. The main idea of FDR is to group a set of mutually exclusive ground atoms (facts) into a variable. This means that only one value can be assigned to the variable at particular time during the execution of the plan. For example, a robot can only be in one place at

[16]Some papers are using another name i.e. the Multi-valued Planning Tasks representation.

the same time. This invariant property can be exploited later by the planner during the search process.

A classical planning problem can be defined in a Finite Domain Representation (FDR) as follows:

Definition 2.2 (**Finite Domain Representation Task** [Helmert, 2009])**.**
A planning problem in finite domain representation (FDR task) is given by 4-tuple $\Pi = \langle V, A, s_0, s_g \rangle$ where:

- $V = \{v_1, ..., v_n\}$ is a set of state variables, each of which is associated with a finite domain D_v. If $d \in D_v$ then we call the pair $v = d$ an atom. A partial variable assignment over V is a function s on some subset of V such that $s(v) \in D_v$, wherever $s(v)$ is defined. If $s(v)$ is defined for all $v \in V$, then s is called a state;
- A is a set of actions, each of which is a 4-tuple $\langle name, cost, pre, eff \rangle$, where *name* is a unique symbol to distinguish an action from others, $cost \in \mathbb{R}^{0+}$ is a non-negative cost, while *pre* and *eff* are partial variable assignments called preconditions and effects respectively;
- s_0 is a state called an initial state and s_g is a partial variable assignment called the goal.

■

An FDR action a is applicable in state s if every atom of the action preconditions is in s. Applying the action a to state s will assign new value to variables as defined in the action effects.

[Helmert, 2009] stated that some planning approaches can get a significant benefit from using FDR. For examples: planners that use decomposition techniques, such as the causal graph heuristic [Helmert, 2004, Helmert and Geffner, 2008] and the multi-agent decomposition heuristic [Crosby, 2014], benefit from a simpler causal structure of the FDR variables; planners that use *landmarks* heuristics [Richter and Westphal, 2010] can compute the landmarks in a more efficient way by exploiting the domain transition graph of the FDR variable domains.

2.2.4 Heuristic Search

As mentioned above, [Bylander, 1994] has shown that in general, solving a planning problem is a PSPACE-complete problem. This makes a naive search approach to

be very inefficient. This fact encourages researchers to develop heuristic search approaches that use particular evaluation function to guide the search. This approach can significantly reduce the search time by exploiting particular structures available in the planning problem in order to compute the appropriate evaluation function. To gain a significant benefit, the evaluation itself must be computed in reasonable time e.g. polynomial-time.

The approaches are based on the *best-first search* technique which is an instance of the general *graph-search* algorithm in which a node is selected for expansion based on evaluation function $f(n)$ [Russell and Norvig, 2009]. The value of the evaluation function is used to determine the node that will be expanded, in this case the node with the *lowest* evaluation will be expanded first.

The choice of f determines the strategy of the search algorithm. If f only has a heuristic function $h(n)$ component, then the algorithm is using *greedy search* strategy, that is: $f(n) = h(n)$. Other strategies may have two components of the evaluation function. For example, A* search strategy is combining the path cost from the start node to node n, $g(n)$, with the heuristic function, that is: $f(n) = g(n) + h(n)$. Every search strategy has its own property. Greedy search is very fast in finding a solution when we have a good heuristic function, but it is also incomplete[17]. A* search is commonly slower than greedy search, but it is complete[18] since it will always backtracks if it finds a dead-end. If we use an *admissible heuristic function*[19] with A* search, then it will be guaranteed that the solution will be globally optimal.

This subsection introduces two techniques for estimating the heuristic $h(n)$, which are used in this thesis. The first is the FastForward heuristic (h^{FF}) that estimates the heuristic by relaxing the planning problem. The second is the Landmarks heuristic (h^{LM}) that estimates the heuristic by computing the number of invariant atoms that should be achieved by the plan.

2.2.4.1 FastForward Heuristic

The FastForward heuristic (h^{FF}) was first introduced by the FastForward (FF) planning system in 2001 [Hoffmann and Nebel, 2001] that has won numerous awards at the International Planning Competition. The computation of h^{FF} is quite simple and only required polynomial-time. It also returns relatively an accurate estimation.

[17] It may not find a solution even though the solution exists.

[18] It will find a solution when the solution does exist.

[19] A heuristic function is admissible if it never overestimates the real cost of the node to reach the goal.

The basic idea of h^{FF} is relaxing the planning problem by ignoring all negative effects when applying an action to the state – a robot can be in two or more places at a time, for example. This basic idea can be applied when we use STRIPS encoding[20]. Since all negative effects are ignored, then the application of the action will increase the number of atoms of a state, which increases the possibility of other actions to be applicable at the next state. Any action can be applied immediately since it does not have any negative consequence.

The heuristic function is calculated using a relaxed *planning graph*[21]. To create a relaxed planning graph, we start from the initial state. Then, we search every applicable action and then add to the graph if it has not been used before. When applying the action, all positive effects of the action are added to the state to produce the next state. This is done repeatedly until no more actions can be added to the graph or the state has satisfied the goal.

To generate an accurate heuristic estimation, the algorithm will extract a relaxed plan from the relaxed planning graph. For each goal position, the algorithm can trace a backward route through the graph to the initial state. The actions that appear in the route are the relaxed plan of the relaxed planning problem. The number of actions in the plan will determine the heuristic value which can help guiding the search.

In FF planning system, the solution plan is found using a greedy forward search from the initial state. An action is selected if it is applicable and it has the lowest heuristic value. If no improved successor can be found, then this greedy search returns failure and the FF planning system will switch to use A* search using the same heuristic calculation. Empirically, the greedy search is good enough to find a solution plan for a large number of existing planning domains.

2.2.4.2 Landmarks Heuristic

[Porteous et al., 2001, Hoffmann et al., 2004] introduced techniques on using *landmarks* for guiding the heuristic search. They define landmarks as facts (propositions) that must be true at some point in every valid solution plan. It was preceded by earlier works such as [Koehler, 1998] that provide hints about the order in which the goals

[20][Helmert, 2006] adapted this idea for FDR encoding by allowing the variables to hold more than one value after applying an action to the state.

[21]Planning graph [Blum and Furst, 1997b] uses a graph structure where nodes correspond to world state propositions and actions, and arcs correspond to preconditions and effects of actions. The algorithms expand the graph from the initial state until reaching the last layer that contains all goals which must not be mutually exclusive. The solution (plan) can be found by applying a backward-search algorithm from the last until reaching the first proposition layer.

should be achieved.

For example, if a robot wants to move from room X to room Y and there is a door between the rooms, then based on the definition, the goal `location(robot,Y)` is a landmark and the fact `doorOpen(X,Y)` is also a landmark. We can also conclude an ordering between these two landmarks, that is: `doorOpen(X,Y)` $\rightarrow$ `location(robot,Y)`. Conceptually, to achieve the goal, the algorithm will first find a partial plan from the initial state to a state that makes true the landmark `doorOpen(X,Y)`, and then find another partial plan from this state to the goal.

The LAMA planning system [Richter and Westphal, 2010] has used landmarks for estimating the heuristic, and it has been proved to be very successful receiving awards in the International Planning Competition. It uses an adapted technique for FDR encoding [Richter et al., 2008] that exploits the structure of the domain transition graph[22](DTG) to efficiently generating the landmarks.

There are several steps to generate the landmarks. First, by definition, every goal is a landmark. This is because the goal must be true in any plan. And then for each goal variable, the algorithm checks the variable's DTG if there is a node (variable value) that occurs on every path from the initial value to the goal value, then this node is also a landmark, which can be naturally ordered before the goal value. After all landmarks have been generated, additional natural orderings are introduced. For all landmark A and B, if A occurs before B in the relaxed planning graph, then an ordering $A \rightarrow B$ is added.

As mentioned above, landmarks can be used as intermediary goals. Instead of searching the goal, the algorithm will iteratively aim to achieve a landmark that is minimal with respect to the orderings using another heuristic technique such as h^{FF} . However, [Hoffmann et al., 2004] observed that the solution plans are often longer than the ones produced by standard h^{FF} , or sometimes it leads to dead ends.

[Richter et al., 2008] introduced a straighforward way to using landmarks to guide the search by counting the number of landmarks that still need to be achieved from particular state onwards. Assume l is this number, it will estimate the goal distance from the state, and it is computed using an equation: $l = n - m + k$, where n is the total number of landmarks, m is the number of accepted landmarks, and k is the number of accepted landmakrs that are required again. Empirically, using this simple landmarks counting in the best-first search leads to good results in some planning domains.

[22]The domain transition graph of a state variable v of an FDR task is the directed graph where nodes are the possible values of v, an arc $\langle d, d' \rangle$ is included in the graph iff $d \neq d'$ and there is an action whose preconditions $pre(v) = d$ or $pre(v)$ undefined, and effects $\mathit{eff}(v) = d'$ [Helmert, 2009].

2.3 Syntax and Semantics

This section provides the backgrounds of the notation and style of the semantics used in §3.

2.3.1 Syntax

The syntax of a language defines the strings of characters which make meaningful statements in the language. This is usually expressed using a context-free grammar such as Backus-Naur Form (BNF). For example:

$$\begin{aligned} \textit{digit} &::= \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9} \\ \textit{number} &::= \textit{digit} \mid \textit{digit number} \end{aligned}$$

Any finite number of a certain element can be listed explicitly. The above equations define two elements. The first is a *digit* element that must be a 0, a 1, ..., or a 9. The second defines a *number* which may have a digit structure, or a digit concatenated with something that has a number structure.

On the other hand, we can use recursion to specify an infinite number of elements (e.g. *number* element above), or we can use the Kleene $(...)^*$ or $(...)^+$ whose meaning are the same as in the regular expression.

The literal strings that appear in the language itself is written using `typewriter` font. Every class of elements is called as *non-terminal* symbols which is written in *italics* fonts.

Studying the semantics of a language usually deal with slightly higher-level concepts representation by some *abstract syntax*, which defines sets of abstract phrases that are independent of any particular representation, but which also provides a simple representation for these phrases [Reynolds, 1998]. This would typically include *terminals* such as *number*, *identifier*, or *string*, as well as *non-terminals* elements without describing their details representation as input strings.

2.3.2 Semantics

The syntax of a language defines expressions that are legal in the language, but it does not specify about the *meaning* of those expressions. A *syntactically correct* configuration specification will be successfully compiled, but the language syntax itself is insufficient to provide information about the configuration that will appear when the specification is deployed.

Thus, the *semantics* of the language [Schmidt, 1997] is required to provide the definitions of the meaning of every element that appears in the specification. For example, although the following strings are written using different syntax, but intuitively they actually have the same meaning (semantics):

```
21
twenty one
XXI
0x15
```

There are several different approaches to specifying the semantics of programming languages. However, declarative configuration languages, such as SmartFrog language, are quite different from most programming (or scripting) languages – they define the characteristics of the intended configuration rather than the *process* of deploying that configuration. In addition, they often emphasise additional features which may not be prominent in a mainstream programming language, for examples: prototypes, value-inheritance, and composition. This motivates our choice of the *denotational* approach to the semantics rather than an *operational* or *axiomatic* approach. This is similar with the approach normally used in other domains such as database query languages.

2.3.3 Denotational Semantics

The denotational semantics defines the meaning of a language by mapping the expressions directly to their meanings called *denotations*, which are usually mathematical values such as a number or a function [Schmidt, 1997]. The basic structure of the semantics involves: the abstract syntax, the semantic algebra, and the valuation functions.

2.3.4 Semantic Algebra

A set of elements that share a common property or use can be grouped together into a *semantic domain*. For example, all numbers can be grouped into a natural numbers domain. Accompanying the domain is a set of *semantic operators*, that operates over the elements of the domain e.g. $\times$ is a binary operator that multiplies two numbers and returns another number. A *semantic domain* together with its *semantic operator* is called *semantic algebra*.

Using the algebra, we can construct expressions in particular domains. This representation can be simplified to produce the simplest representation as the final result. The simplification itself should preserve the underlying meaning of the expression.

2.3.5 Valuation Functions

A *valuation function* defines how to map the elements of the syntax into their meaning (a denotation). For example[23]:

$$\begin{aligned} \mathbf{F}[\![\texttt{21}]\!] &:= 21 \\ \mathbf{F}[\![\texttt{twenty one}]\!] &:= 21 \\ \mathbf{F}[\![\texttt{XXI}]\!] &:= 21 \\ \mathbf{F}[\![\texttt{0x15}]\!] &:= 21 \end{aligned}$$

Note that the left hand side elements (`typewriter` fonts) are literal strings which appear in the source language, while the right hand side elements (roman fonts) are abstract numbers. The functions show that the meaning of all elements are the same: `21` is the same as `twenty one`, and so on.

We can use a similar way in recursive syntax to define the semantics of literal strings of arbitrary length. For example, the following equations define the meanings of a binary numbers language:

$$\begin{aligned} \mathbf{F}[\![\texttt{0}]\!] &:= 0 \\ \mathbf{F}[\![\texttt{1}]\!] &:= 1 \\ \mathbf{F}[\![N\ \texttt{0}]\!] &:= 2 \times \mathbf{F}[\![N]\!] \\ \mathbf{F}[\![N\ \texttt{1}]\!] &:= 2 \times \mathbf{F}[\![N]\!] + 1 \end{aligned}$$

Note that N is a string literal of binary digits (`0` or `1`), and $\mathbf{F}[\![N]\!]$ is the denotation of N (the number represented by the string literal N).

By extending the semantics, we can create similar expressions for the other representations. For example, the following two expressions have different syntaxes:

$$\begin{aligned} \mathit{expr} &:= \mathit{number} \mid (\texttt{mul}\ \mathit{expr}\ \mathit{expr}) \\ \mathit{expr} &:= \mathit{number} \mid \mathit{expr}\ \texttt{*}\ \mathit{expr} \end{aligned}$$

but they have the same semantics:

$$\begin{aligned} \mathbf{F}[\![(\texttt{mul}\ \mathit{expr}\ \mathit{expr})]\!] &:= \mathbf{F}[\![\mathit{expr}]\!] \times \mathbf{F}[\![\mathit{expr}]\!] \\ \mathbf{F}[\![\mathit{expr}\ \texttt{*}\ \mathit{expr}]\!] &:= \mathbf{F}[\![\mathit{expr}]\!] \times \mathbf{F}[\![\mathit{expr}]\!] \end{aligned}$$

Note that `*` on left-hand side is a string literal of the source language, while $\times$ is the abstract mathematical multiplication operator.

[23] := is read as "is define as".

2.4 Summary

Unlike imperative ones, practical declarative configuration tools guarantee that the final state of the system would match to our requirements since it is explicitly defined in the specification. But they cannot guarantee that the ordering constraints of configuration changes are maintained across multiple machines. Solutions offered by some tools, e.g. SmartFrog (Behavioural Signature model) and CDDLM (control-flow), cannot completely solve this issue because the ordering constraints must be manually defined. This is conceptually similar to imperative scripts.

Some previous works have shown that the automated planner is a viable solution to the above problem – the planner automatically generates a workflow that can bring the system to the desired state from an arbitrary state while preserving the ordering constraints. However, we are not aware of previous works that allows us to define global constraints as part of the goals. On the other hand, we cannot find any practical declarative language that has notions of actions and global constraints which are required in the planning process. This makes difficult for practical declarative tools to make use of the planner.

Some configuration tools have been implementing a weakly distributed architecture to avoid bottleneck and single point of failure issues, which is commonly exist in a centralised system. This type of architecture can also retain some degree of centralised control where the fully-distributed cannot provide. We are not aware of previous works of dynamic-workflow approach that has been using a weakly distributed architecture.

The planning problem for reconfiguring cloud systems are *non-deterministic* and *dynamic*. Although the problem cannot satisfy several assumptions of classical planning, but we can still use a classical planner as part of the framework to solve the problem, in particular for generating a plan to bring the system from current to the desired state.

Chapter 3

Modelling Configuration Changes

The configuration specification defined in a particular language plays an important role in every configuration tool because it drives any change during deployment. Any misunderstanding by the user or misinterpretation by the tool on the specification will yield undesired result that can threaten the operation of the system. Thus, it is necessary to have a clear formal semantics of the configuration language which is independent from the implementation. This will give us some benefits:

- The formal semantics acts as a precise, independent reference for developers, so that we can confidently create different implementations with the same behaviours.
- It is easier to create interoperable tools for generating, analysing, refactoring, and visualising configurations, for example: automatically analysing the *provenance* of configurations for fault and security analysis.
- The process of formalising the semantics can expose ambiguities and other issues with the language itself.
- It is easier to extend the language since every element has a precise definition and meaning.

We are not aware of any document that provides the formal semantics of the declarative configuration languages that are used in real ”production“ system such as the SmartFrog language [Goldsack et al., 2009].

On the other hand, we have not found any declarative configuration language that has the notions of actions and the global constraints. With the absence of these notions, the automated planner cannot use the configuration specification for computing

sound workflows between any two viable states. Using a separated language such as PDDL for describing the actions and the global constraints is not a good solution because it lacks desired features which are available in practical configuration languages such as object-oriented modelling, inheritance, and composition. In addition, a mixed representation is difficult to be accepted because the administrators have to learn two languages at the same time, where both have different semantics – this is clearly error-prone.

This thesis describe a work to extend the SmartFrog (SF) language for introducing the notions of actions and the global constraints. However, introducing new notations in the language is not an easy task since this could introduce ambiguities to the language itself. Thus, we first formalised the SF semantics, and then used this formal semantics as the foundation to extend the language.

This chapter describes two contributions of this thesis. The first is the formal semantics of the core subset of SmartFrog (SF) language using denotational semantics approach. The second is extending the SF language to introduce a static-type system and new notations for describing actions and global constraints. This extention is called SFP language which will then be used in the rest of this thesis.

This chapter is organised as follows. The first section provides a brief introduction of SF language by examples. It is then followed by a section that describes the formal semantic of the core subset of SF language. The next section provides a brief introduction of SFP language by examples. Finally, the formal semantic of SFP language is described in the last section.

3.1 SmartFrog by Example

SmartFrog (SF) language is a dynamic-typed prototype based language for specifying the declarative configuration state of a system. The configuration is described as a collection of variables, each of which can be assigned with a primitive value i.e. boolean, numeric, and string, a reference, a vector or a component – an SF component is similar to an object. A component can have a set of attributes that could be assigned with particular (untyped) value. Several components can be composed in a tree-like structure by assigning a component to an attribute of another component. The following example provides a brief introduction on how to specify the configuration state of a system in SF.

Assume that we want to have a system whose desired state is depicted in figure 3.1.

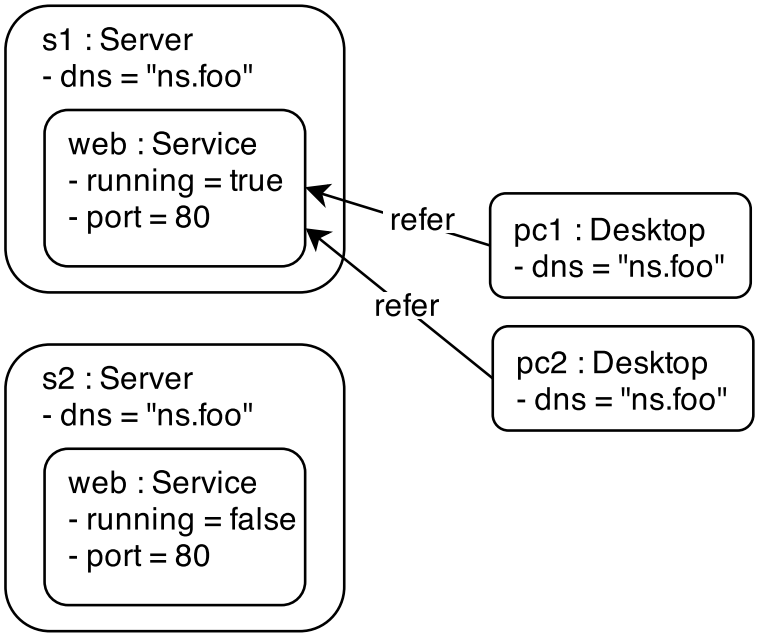


Figure 3.1: An example system that consists of two web servers and two clients.

It consists of four machines: server s_1 and s_2, and client pc_1 and pc_2. Each machine has attribute *dns* that holds the address of the DNS server. Each server has service *web* that serves at port 80 and can be at state *running* or *stopped*. Both pc_1 and pc_2 are referring to service *web* in s_1.

The best way to model this system in SF is using the notion of prototyping where an object can be a prototype of another object. This enables code reuse through inheritance as well as composition where the value of every inherited attributes could be overridden with any specific value. On the other side, SF has a notion of reference that can represent the dependency of one to another component.

Listing 3.1 shows the specification of the example system in SF. Lines 1-3 are the descriptions of a prototype component named `Machine` with attribute `dns` which holds the address of the DNS server. Lines 4-7 describe another prototype component named `Service` with two attributes: `running` and `port` that hold the service's state and port number respectively. Lines 8-19 specify of the main component named `sfConfig`, which holds the final descriptions of the system. Lines 9-11 describe server `s1` that uses `Machine` as the prototype. In addition to `dns` (inherited from `Machine`), it has attribute `web` which represents a service running on server `s1`. Lines 12-14 describe server `s2` that uses `s1` as prototype where the inherited value of `web.running` (`true`) is overridden with `false`. Lines 15-17 specify client `pc1` which uses `Machine` as the prototype. It has an additional attribute named `refer` that represents the reference to

Listing 3.1: The SF specification of the system in figure 3.1.

```
1  Machine extends {
2    dns "ns.foo";
3  }
4  Service extends {
5    running true;
6    port 80;
7  }
8  sfConfig extends {
9    s1 extends Machine {
10     web extends Service;
11   }
12   s2 extends s1 {
13     web.running false;
14   }
15   pc1 extends Machine {
16     refer DATA s1:web;
17   }
18   pc2 pc1;
19 }
```

the service on server `s1`. Since keyword **DATA** precedes the reference, then it is kept as the value and not resolved – this type of reference is called as *data reference*. Line 18 defines another client `pc2` that is assigned with a reference of `pc1`. Since it is not preceded by keyword **DATA**, then the reference is resolved by the compiler where the dereference value is copied and assigned to `pc1` – this type of reference is called as *link reference*. Finally, every variable defined inside main component `sfConfig` is presented as the final specification. Figure 3.2 shows the output from the compiler presented in YAML [yam, 2014], which is an evaluation of the main `sfConfig` object.

The above example covers most of the SF core features i.e. prototyping, component (object), primitive value, data reference, link reference, and main component (`sfConfig`). Vector is not shown in the example. It is similar to the standard list data structure where you can create a list of any basic value (primitive, data reference, or vector). For example:

```
1 sfConfig extends {
2   myvector [ true, 1, "a␣string", [ false, 0 ] ];
3 }
```

The next section defines the formal semantic of the core features using denotational semantic.

```
s1:
  dns: "ns.foo"
  web:
    running: true
    port: 80
s2:
  dns: "ns.foo"
  web:
    running: false
    port: 80
pc1:
  dns: "ns.foo"
  refer: s1:web
pc2:
  dns: "ns.foo"
  refer: s1:web
```

Figure 3.2: The compiler output of SF specification in listing 3.1 in YAML.

3.2 Formal Semantic of SmartFrog Language

The SF semantics consists of three parts: the abstract syntax (§3.2.1), the semantic algebras (§3.2.2), and the valuation functions (§3.2.3). It is using the notations described in §2.3.

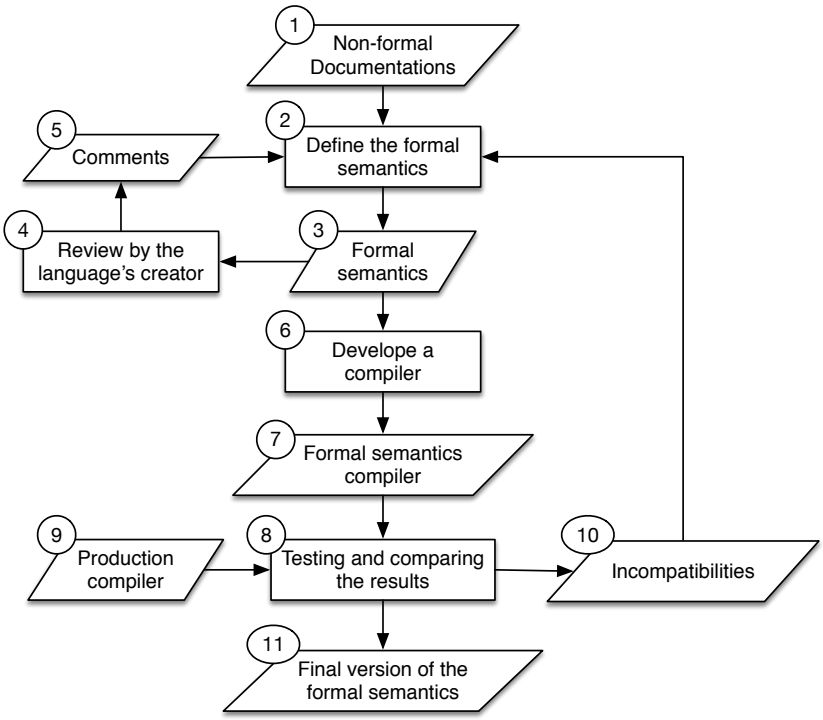


Figure 3.3: The process for developing the formal semantics of SmartFrog language.

Figure 3.3 shows the process used for developing all parts of this semantics, which can be described in details as follows[1]:

- First, we used (1) a non-formal documentation of the SmartFrog language to (2) define the semantics.
- This (3) formal semantics was then given to (4) the language's creator to be reviewed, and then we refine the semantics based on (5) his feedbacks (comments).
- For further validation, the formal semantics was used as (6) a single reference for developers to develop (7) the compilers. These compilers were then (8) tested using: manually written specifications containing edge-cases, example specifications in the SmartFrog software distribution, and specifications generated by an automated program. The test outputs of the compilers were compared againts (9) the existing production compiler in order to find any (10) incompatibility. Any finding will then be used to refine the semantics.
- Finally, after the creator has agreed that the formal semantics is correct, and the compilers developed from the semantics were producing outputs which are compatible with the production compiler, then (11) the final version of the semantics was released.

3.2.1 Abtract Syntax

Definition 3.1 (**SF Terminal Symbols**)**.** These are the basic symbols of the language which appear in the source code:

$$\begin{aligned} \text{Bool} &\in \textit{Boolean} \\ \text{Num} &\in \textit{Number} \\ \text{Str} &\in \textit{String} \\ \text{Null} &\in \textit{NullValue} \\ \text{I} &\in \textit{Identifier} \end{aligned}$$

■

Definition 3.2 (**SF Non Terminal Symbols**)**.** These are the non-terminal elements of the syntax:

[1] Each number represents the component in figure 3.3

$$
\begin{aligned}
\text{SF} &\in \textit{SFSpecification} \\
\text{B} &\in \textit{Block} \\
\text{A} &\in \textit{Assignment} \\
\text{P} &\in \textit{Prototype} \\
\text{V} &\in \textit{Value} \\
\text{R} &\in \textit{Reference} \\
\text{DR} &\in \textit{DataReference} \\
\text{LR} &\in \textit{LinkReference} \\
\text{Vec} &\in \textit{Vector} \\
\text{BV} &\in \textit{BasicValue}
\end{aligned}
$$

■

Definition 3.3 (SF Abstract Syntax). The non-terminals are defined by the following *abstract syntax*. Note that the syntax does not specify the details of the concrete syntax – for example, an assignment includes a *reference* and a *value*, but we do not care about the punctuation or the keywords which are used to represent this in the source code (see appendix A.1 for the SF concrete syntax).

$$
\begin{aligned}
\text{SF} &::= \text{B} \\
\text{B} &::= \text{A B} \mid \varepsilon \\
\text{A} &::= \text{R V} \\
\text{V} &::= \text{BV} \mid \text{LR} \mid \text{P} \\
\text{P} &::= \text{R P} \mid \text{B P} \mid \varepsilon \\
\text{BV} &::= \text{Bool} \mid \text{Num} \mid \text{Str} \mid \text{Null} \mid \text{Vec} \mid \text{DR} \\
\text{LR} &::= \text{R} \\
\text{DR} &::= \text{R} \\
\text{Vec} &::= (\text{BV})^* \\
\text{R} &::= \text{I} \mid \text{I R}
\end{aligned}
$$

■

The symbol ε represents the empty string, and $(\text{L})^*$ and $(\text{L})^+$ represent possibly-empty and non-empty lists of elements of type L respectively.

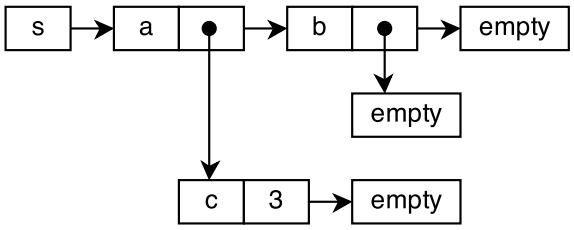
3.2.2 Semantic Algebras

3.2.2.1 Semantics Domains

The *primary semantic domain* is used to represent the output of the compilation process – since SF is a declarative configuration language, then this output represents the configuration that will appear when the specification is deployed. For the SF compiler, this domain is a tree of attribute-value pairs. For example, after compilation, the following SF specification:

```
1 a extends {
2   c 3;
3 }
4 b extends {
5 }
```

would be represented as follows:



Definition 3.4 (**List Data Structure**)**.** Elements of $(L)^*$ are lists of L-elements: $l_1 :: l_2 :: ... :: l_n :: \varnothing_L$, $n \geq 0$, and $\varnothing_L$ represents an empty list. If $x = l_1 :: l_2 :: ... :: l_n :: \varnothing_L$, then $|x| = n$ and $l' \in x$ iff $\exists i \,.\, l' = l_i$. ■

We use the *store* domain as the primary domain. This domain models a computer memory as a tree of variable-value pairs.

Definition 3.5 (**The primary domain**)**.** Formally, the *Store* domain $\mathcal{S} = (\mathbb{I} \times \mathcal{V})^*$, is a list of identifier-value pairs, where $\mathcal{V} = \mathbb{V} \cup \mathcal{S}$. We use $\varnothing_\mathcal{S}$ to represent an empty store, and $\mathcal{V}_\perp = \mathcal{V} \cup \{\perp\}$ for a lifted primary domain which includes the undefined value $\perp$. ■

Definition 3.6 (**The secondary domains**)**.** The values of the attributes belong to one of these secondary domains:
$\mathbb{I}$ is the identifier domain;
$\mathbb{B} = \{True, False\}$, is the boolean domain;
$\mathcal{N} = \{Null\}$, the null domain;
$\mathbb{R}$ is the real number domain;

$\mathbb{S}$ is the string domain;
$\mathcal{R} = \{r_i \mid r_i \in (\mathbb{I})^*\}$, is the reference domain where $\varnothing_{\mathbb{I}}$ represents an empty reference;
$\mathcal{L} = \{l_i \mid l_i = \langle link, r_i \rangle, r_i \in \mathcal{R}\}$, is the link reference domain where *link* is a tag symbol;
$\mathcal{B} = \mathbb{B} \uplus \mathbb{R} \uplus \mathbb{S} \uplus \mathcal{R} \uplus \mathcal{N} \uplus \{v_i \mid v_i \in (\mathcal{B})^*\}$, is the basic value domain;
$\mathbb{C}$ is the constraint domain;
$\mathbb{A}$ is the action domain;
$\mathbb{V} = \mathcal{B} \uplus \mathcal{L} \uplus \mathbb{C} \uplus \mathbb{A}$, is the value domain. ■

Note that $\mathbb{C}$ and $\mathbb{A}$ are not used in this semantics. The purpose of including them in the definition is solely because the domains and their operators and functions will be reused later in section 3.4.

Notice that a *reference* is the sequence of identifiers specifying the "path" to a particular attribute in the tree structure. An empty reference ($\varnothing_{\mathbb{I}}$) refers to the root. A reference can be written as a concatenation of an identifier with another reference, for example: $r := id :: r'$, where $id \in \mathbb{I}$ and $r, r' \in \mathcal{R}$. On the other hand, vectors may be nested, and in particular a single-element vector is not the same as the element itself.

The above example specification can be represented using store s as follows:

$$s = \langle \mathtt{a}, \langle \mathtt{c}, 3 \rangle :: \varnothing_{\mathcal{S}} \rangle :: \langle \mathtt{b}, \varnothing_{\mathcal{S}} \rangle :: \varnothing_{\mathcal{S}}$$

3.2.2.2 Semantic Operations

This section defines the fundamental operations on the semantic domains. These are analogous to the basic arithmetic operations in the examples from §2.3.2 – in this case, they operate on the above primary and secondary domains rather than natural numbers.

Definition 3.7 (Operator $\oplus$).
$\oplus$ is a binary operator that returns the concatenation of the operands. Either operand may be an identifier or a reference, but the result is always a reference.

$$\begin{array}{rcl} \oplus : \mathbb{I} \times \mathbb{I} & \to & \mathcal{R} \\ id_1 \oplus id_2 & := & id_1 :: id_2 :: \varnothing_{\mathbb{I}} \end{array} \qquad \begin{array}{rcl} \oplus : \mathbb{I} \times \mathcal{R} & \to & \mathcal{R} \\ id \oplus r & := & id :: r \end{array}$$

$$\begin{array}{rcl} \oplus : \mathcal{R} \times \mathcal{R} & \to & \mathcal{R} \\ \varnothing_{\mathbb{I}} \oplus r & := & r \\ (id :: r) \oplus \varnothing_{\mathbb{I}} & := & id :: r \\ (id :: r_1) \oplus r_2 & := & id :: (r_1 \oplus r_2) \end{array} \qquad \begin{array}{rcl} \oplus : \mathcal{R} \times \mathbb{I} & \to & \mathcal{R} \\ \varnothing_{\mathbb{I}} \oplus id & := & id :: \varnothing_{\mathbb{I}} \\ (id :: r) \oplus id & := & id :: (r \oplus id) \end{array}$$

Where :: is the concatenation operator which prepends a value to a list, r is a list of identifiers, and $\varnothing_{\mathbb{I}}$ is the empty list. ■

Example.

$$\begin{aligned}
\mathtt{a} \oplus \mathtt{b} &= \mathtt{a} :: \mathtt{b} :: \varnothing_{\mathbb{I}} \\
\mathtt{a} \oplus \mathtt{b} :: \mathtt{c} :: \varnothing_{\mathbb{I}} &= \mathtt{a} :: \mathtt{b} :: \mathtt{c} :: \varnothing_{\mathbb{I}} \\
\mathtt{b} :: \mathtt{c} :: \varnothing_{\mathbb{I}} \oplus \mathtt{a} &= \mathtt{b} :: \mathtt{c} :: \mathtt{a} :: \varnothing_{\mathbb{I}} \\
\mathtt{a} :: \mathtt{b} :: \varnothing_{\mathbb{I}} \oplus \mathtt{c} :: \mathtt{d} :: \varnothing_{\mathbb{I}} &= \mathtt{a} :: \mathtt{b} :: \mathtt{c} :: \mathtt{d} :: \varnothing_{\mathbb{I}}
\end{aligned}$$

Definition 3.8 (Operator $\ominus$).

$\ominus$ is a binary operator that returns the first operand with any common prefix removed.

$$\ominus : \mathcal{R} \times \mathcal{R} \to \mathcal{R}$$

$$\begin{aligned}
\varnothing_{\mathbb{I}} \ominus r &:= \varnothing_{\mathbb{I}} & r \ominus \varnothing_{\mathbb{I}} &:= r \\
(id :: r_1) \ominus (id :: r_2) &:= r_1 \ominus r_2 & (id_1 :: r_1) \ominus (id_2 :: r_2) &:= id_1 \ominus r_1
\end{aligned}$$

■

Example.

$$\begin{aligned}
\mathtt{a} :: \mathtt{b} :: \varnothing_{\mathbb{I}} \ominus \mathtt{a} :: \mathtt{b} :: \varnothing_{\mathbb{I}} &= \varnothing_{\mathbb{I}} \\
\mathtt{a} :: \mathtt{b} :: \varnothing_{\mathbb{I}} \ominus \mathtt{a} :: \mathtt{b} :: \mathtt{c} :: \varnothing_{\mathbb{I}} &= \varnothing_{\mathbb{I}} \\
\mathtt{a} :: \mathtt{b} :: \mathtt{c} :: \varnothing_{\mathbb{I}} \ominus \mathtt{a} :: \mathtt{b} :: \varnothing_{\mathbb{I}} &= \mathtt{c} :: \varnothing_{\mathbb{I}} \\
\mathtt{a} :: \mathtt{b} :: \mathtt{c} :: \varnothing_{\mathbb{I}} \ominus \mathtt{a} :: \mathtt{b} :: \mathtt{d} :: \varnothing_{\mathbb{I}} &= \mathtt{c} :: \varnothing_{\mathbb{I}}
\end{aligned}$$

Definition 3.9 (Operator $\equiv$).

$\equiv$ is a binary operator that returns *True* if two references are equal, otherwise it returns *False*.

$$\equiv \; : \; \mathcal{R} \times \mathcal{R} \to \mathbb{B}$$

$$\begin{aligned}
\varnothing_{\mathbb{I}} \equiv \varnothing_{\mathbb{I}} &:= \mathit{True} \\
id :: rest \equiv \varnothing_{\mathbb{I}} &:= \mathit{False} \\
\varnothing_{\mathbb{I}} \equiv id :: rest &:= \mathit{False} \\
id_1 :: rest_1 \equiv id_2 :: rest_2 &:= (id_1 = id_2) \wedge (rest_1 \equiv rest_2)
\end{aligned}$$

■

Example.

$$\begin{aligned}
\mathtt{a} :: b :: \varnothing_{\mathbb{I}} \equiv \mathtt{a} :: b :: \varnothing_{\mathbb{I}} &= \mathit{True} & \mathtt{a} :: b :: \varnothing_{\mathbb{I}} \equiv \mathtt{a} :: c :: \varnothing_{\mathbb{I}} &= \mathit{False} \\
\mathtt{a} :: b :: \varnothing_{\mathbb{I}} \equiv \mathtt{a} :: \varnothing_{\mathbb{I}} &= \mathit{False}
\end{aligned}$$

Definition 3.10 (Operator $\subseteq_{\mathcal{R}}$ and $\subset_{\mathcal{R}}$).

$\subseteq_{\mathcal{R}}$ and $\subset_{\mathcal{R}}$ are true if the left reference is a strict or non-strict prefix of the right one.

$\subseteq_{\mathcal{R}} : \mathcal{R} \times \mathcal{R} \to \mathbb{B}$ $\quad\quad$ $\subset_{\mathcal{R}} : \mathcal{R} \times \mathcal{R} \to \mathbb{B}$

$r_1 \subseteq_{\mathcal{R}} r_2 \;:=\; ((r_1 \ominus r_2) = \varnothing_{\mathbb{I}})$ $\quad\quad$ $r_1 \subset r_2 \;:=\; (r_1 \subseteq r_2) \wedge \neg(r_1 \equiv r_2)$

■

Example.

$\mathtt{a} :: \mathtt{b} :: \varnothing_{\mathbb{I}} \subseteq_{\mathcal{R}} \mathtt{a} :: \mathtt{b} :: \varnothing_{\mathbb{I}} = \mathit{True}$ $\quad\quad$ $\mathtt{a} :: \mathtt{b} :: \varnothing_{\mathbb{I}} \subset_{\mathcal{R}} \mathtt{a} :: \mathtt{b} :: \varnothing_{\mathbb{I}} = \mathit{False}$

$\mathtt{a} :: \mathtt{b} :: \varnothing_{\mathbb{I}} \subseteq_{\mathcal{R}} \mathtt{a} :: \mathtt{b} :: \mathtt{c} :: \varnothing_{\mathbb{I}} = \mathit{True}$ $\quad\quad$ $\mathtt{a} :: \mathtt{b} :: \varnothing_{\mathbb{I}} \subset_{\mathcal{R}} \mathtt{a} :: \mathtt{b} :: \mathtt{c} :: \varnothing_{\mathbb{I}} = \mathit{True}$

$\mathtt{a} :: \mathtt{b} :: \mathtt{d} :: \varnothing_{\mathbb{I}} \subseteq_{\mathcal{R}} \mathtt{a} :: \mathtt{b} :: \mathtt{c} :: \varnothing_{\mathbb{I}} = \mathit{False}$

$\mathtt{a} :: \mathtt{b} :: \mathtt{d} :: \varnothing_{\mathbb{I}} \subset_{\mathcal{R}} \mathtt{a} :: \mathtt{b} :: \mathtt{c} :: \varnothing_{\mathbb{I}} = \mathit{False}$

Definition 3.11 (*prefix*).

The ***prefix*** function returns the longest strict prefix of the given reference.

$\mathit{prefix} : \mathcal{R} \to \mathcal{R}$

$$\begin{aligned} \mathit{prefix}(\varnothing_{\mathbb{I}}) &:= \varnothing_{\mathbb{I}} \\ \mathit{prefix}(id :: \varnothing_{\mathbb{I}}) &:= \varnothing_{\mathbb{I}} \quad , \text{where} id \in \mathbb{I} \\ \mathit{prefix}(r :: id :: \varnothing_{\mathbb{I}}) &:= r :: \varnothing_{\mathbb{I}} \quad , \text{where} r \in \mathcal{R} \end{aligned}$$

■

Example.

$\mathit{prefix}(\mathtt{a} :: \varnothing_{\mathbb{I}}) = \varnothing_{\mathbb{I}}$ $\quad\quad$ $\mathit{prefix}(\mathtt{a} :: \mathtt{b} :: \mathtt{c} :: \varnothing_{\mathbb{I}}) = \mathtt{a} :: \mathtt{b} :: \varnothing_{\mathbb{I}}$

Definition 3.12 (*put*).

The ***put*** function updates the value of an identifier in a store, or adds it if it does not already exist. Notice that this operates only on single identifier – the following function (bind) extends this to support hierarchical references.

$\mathit{put} : \mathcal{S} \times \mathbb{I} \times \mathcal{V} \to \mathcal{S}$

$$\begin{aligned} \mathit{put}(\varnothing_{\mathcal{S}}, id, v) &:= \langle id, v \rangle :: \varnothing_{\mathcal{S}} \\ \mathit{put}(\langle id, v_s \rangle :: s_p, id, v) &:= \langle id, v \rangle :: s_p \\ \mathit{put}(\langle id_s, v_s \rangle :: s_p, id, v) &:= \langle id_s, v_s \rangle :: \mathit{put}(s_p, id, v) \end{aligned}$$

■

Example.

$$\begin{aligned} \mathit{put}(\varnothing_{\mathcal{S}}, \mathtt{a}, 1) &= \langle \mathtt{a}, 1 \rangle :: \varnothing_{\mathcal{S}} \\ \mathit{put}(\langle \mathtt{a}, 1 \rangle :: \varnothing_{\mathcal{S}}, \mathtt{b}, 2) &= \langle \mathtt{a}, 1 \rangle :: \langle \mathtt{b}, 2 \rangle :: \varnothing_{\mathcal{S}} \\ \mathit{put}(\langle \mathtt{a}, 1 \rangle :: \langle \mathtt{b}, 2 \rangle :: \varnothing_{\mathcal{S}}, \mathtt{b}, 3) &= \langle \mathtt{a}, 1 \rangle :: \langle \mathtt{b}, 3 \rangle :: \varnothing_{\mathcal{S}} \end{aligned}$$

The following proposition shows that function *put* can maintain the unique identifiers property of the store after operation.

Proposition 3.13. *Assume $s \in \mathcal{S}$ has unique identifiers i.e. $\forall \langle id_i, v_i \rangle, \langle id_j, v_j \rangle \in s \;.\; i \neq j \Rightarrow id_i \neq id_j$. Then operation $s' = put(s, id, v)$ always returns $s' \in \mathcal{S}$ that also has unique identifiers i.e. $\forall \langle id'_i, v'_i \rangle, \langle id'_j, v'_j \rangle \in s' \;.\; i \neq j \Rightarrow id'_i \neq id'_j$.*

Proof. See appendix A.2 □

Definition 3.14 (*bind*).

The ***bind*** function updates the value of a reference in a store. An error[2] occurs if an attempt is made to update a reference whose parent does not exist ($\boldsymbol{err}_2$), or whose parent is not itself a store ($\boldsymbol{err}_1$). It is also illegal to replace the root store ($\boldsymbol{err}_3$).

$bind : \mathcal{S} \times \mathcal{R} \times \mathcal{V} \to \mathcal{S}$

$$
\begin{array}{rcl}
bind(s, \varnothing_{\mathbb{I}}, v) & := & \boldsymbol{err}_3 \\
bind(s, id :: \varnothing_{\mathbb{I}}, v) & := & put(s, id, v) \\
bind(\varnothing_{\mathcal{S}}, id :: r, v) & := & \boldsymbol{err}_2 \\
bind(\langle id, v_s \rangle :: s_p, id :: r, v) & := & \text{if } v_s \in \mathcal{S} \text{ then } \langle id, bind(v_s, r, v) \rangle :: s_p \text{ else } \boldsymbol{err}_1 \\
bind(\langle id_s, v_s \rangle :: s_p, id :: r, v) & := & \langle id_s, v_s \rangle :: bind(s_p, id :: r, v)
\end{array}
$$

■

Example.

$$
\begin{array}{rcl}
bind(\varnothing_{\mathcal{S}}, \mathtt{a} :: \varnothing_{\mathbb{I}}, 1) & = & \langle \mathtt{a}, 1 \rangle :: \varnothing_{\mathcal{S}} \\
bind(\langle \mathtt{a}, 1 \rangle :: \varnothing_{\mathcal{S}}, \mathtt{b} :: \varnothing_{\mathbb{I}}, 2) & = & \langle \mathtt{a}, 1 \rangle :: \langle \mathtt{b}, 2 \rangle :: \varnothing_{\mathcal{S}} \\
bind(\langle \mathtt{a}, 1 \rangle :: \langle \mathtt{b}, 2 \rangle :: \varnothing_{\mathcal{S}}, \mathtt{b} :: \varnothing_{\mathbb{I}}, \varnothing_{\mathcal{S}}) & = & \langle \mathtt{a}, 1 \rangle :: \langle \mathtt{b}, \varnothing_{\mathcal{S}} \rangle :: \varnothing_{\mathcal{S}} \\
bind(\langle \mathtt{a}, 1 \rangle :: \langle \mathtt{b}, 2 \rangle :: \varnothing_{\mathcal{S}}, \mathtt{b} :: \mathtt{c} :: \varnothing_{\mathbb{I}}, 3) & = & \boldsymbol{err}_1 \\
bind(\langle \mathtt{a}, 1 \rangle :: \langle \mathtt{b}, \varnothing_{\mathcal{S}} \rangle :: \varnothing_{\mathcal{S}}, \mathtt{b} :: \mathtt{c} :: \varnothing_{\mathbb{I}}, 3) & = & \langle \mathtt{a}, 1 \rangle :: \langle \mathtt{b}, \langle c, 3 \rangle :: \varnothing_{\mathcal{S}} \rangle :: \varnothing_{\mathcal{S}}
\end{array}
$$

Definition 3.15 (*find*).

The ***find*** function looks up the value of a reference in a store.

$find : \mathcal{S} \times \mathcal{R} \to \mathcal{V}_{\perp}$

$$
\begin{array}{rcl}
find(s, \varnothing_{\mathbb{I}}) & := & s \\
find(\varnothing_{\mathcal{S}}, r) & := & \perp \\
find(\langle id, v_s \rangle :: s', id :: \varnothing_{\mathbb{I}}) & := & v_s \\
find(\langle id_s, v_s \rangle :: s', id :: \varnothing_{\mathbb{I}}) & := & find(s', id :: \varnothing_{\mathbb{I}}) \\
find(\langle id, v_s \rangle :: s', id :: r') & := & \text{if } v_s \in \mathcal{S} \text{ then } find(v_s, r') \text{ else } \perp \\
find(\langle id_s, v_s \rangle :: s', id :: r') & := & find(s', id :: r')
\end{array}
$$

■

[2] The handling of errors in these definitions is rather informal – functions which potentially cause errors are only partially defined. This is not generally desirable in a denotational semantics and could be avoided by using, for example monads. However, the additional complication is not appropriate here.

Example.

$$\begin{aligned}
\mathit{find}(\langle \mathtt{a}, 1\rangle :: \langle \mathtt{b}, \langle c, 3\rangle :: \varnothing_{\mathcal{S}}\rangle :: \varnothing_{\mathcal{S}}, \varnothing_{\mathbb{I}}) &= \langle \mathtt{a}, 1\rangle :: \langle \mathtt{b}, \langle c, 3\rangle :: \varnothing_{\mathcal{S}}\rangle :: \varnothing_{\mathcal{S}} \\
\mathit{find}(\langle \mathtt{a}, 1\rangle :: \langle \mathtt{b}, \langle c, 3\rangle :: \varnothing_{\mathcal{S}}\rangle :: \varnothing_{\mathcal{S}}, \mathtt{a} :: \varnothing_{\mathbb{I}}) &= 1 \\
\mathit{find}(\langle \mathtt{a}, 1\rangle :: \langle \mathtt{b}, \langle c, 3\rangle :: \varnothing_{\mathcal{S}}\rangle :: \varnothing_{\mathcal{S}}, \mathtt{b} :: \varnothing_{\mathbb{I}}) &= \langle c, 3\rangle :: \varnothing_{\mathcal{S}} \\
\mathit{find}(\langle \mathtt{a}, 1\rangle :: \langle \mathtt{b}, \langle c, 3\rangle :: \varnothing_{\mathcal{S}}\rangle :: \varnothing_{\mathcal{S}}, \mathtt{c} :: \varnothing_{\mathbb{I}}) &= \bot \\
\mathit{find}(\langle \mathtt{a}, 1\rangle :: \langle \mathtt{b}, \langle c, 3\rangle :: \varnothing_{\mathcal{S}}\rangle :: \varnothing_{\mathcal{S}}, \mathtt{b} :: \mathtt{c} :: \varnothing_{\mathbb{I}}) &= 3
\end{aligned}$$

Based on definition 3.15, we can define a property of the store's structure: every element's parent is a store, as follows:

Proposition 3.16. *Assume $s \in \mathcal{S}$ then $\forall r \in \mathcal{R}\,.\,\mathit{find}(s,r) \neq \bot \Rightarrow \mathit{find}(s,\mathit{prefix}(r)) \in \mathcal{S}$.*

Proof. See appendix A.2 □

Definition 3.17 (operator $\subset_{\mathcal{S}}$).
$\subset_{\mathcal{S}}$ is true if the left store is a sub-store of the right one.
$\subset_{\mathcal{S}}: \mathcal{S} \times \mathcal{R} \to \mathbb{B}$

$$s_1 \subset_{\mathcal{S}} s_2 \quad := \quad \exists r \in \mathcal{R} \text{ where } \mathit{find}(s_2, r) = s_1 \text{ and } r \neq \varnothing_{\mathbb{I}}$$ ■

Based on definition 3.17, we could define a property of *bind* (similar with proposition 3.13): the *bind* function maintains the uniqueness of identifiers, as follows:

Proposition 3.18. *Assume $s \in \mathcal{S}$ where s has unique identifiers, and $\forall s_i \subset_{\mathcal{S}} s : s_i$ has unique identifiers. Then operation $s' = \mathit{bind}(s, id :: r, v)$ always returns s' that has unique identifiers and $\forall s_j \subset_{\mathcal{S}} s' : s_j$ has unique identifiers as well.*

Proof. See appendix A.2 □

Definition 3.19 (*resolve*).
The ***resolve*** function looks up a reference in a store, by starting with a given *namespace*[3] (reference of the sub-store) and searching up the hierarchy of parent stores until a value is found (or not). It returns a tuple $\langle ns, v\rangle$, where ns is the namespace in which the target element is found and v is the value. If the target is not found then $\langle ns, v\rangle = \langle \varnothing_{\mathbb{I}}, \bot\rangle$.
$\mathit{resolve} : \mathcal{S} \times \mathcal{R} \times \mathcal{R} \to \mathcal{R} \times \mathcal{V}$

$$\begin{aligned}
\mathit{resolve}(s, \varnothing_{\mathbb{I}}, r) &:= \langle \varnothing_{\mathbb{I}},\ \mathit{find}(s,r)\rangle \\
\mathit{resolve}(s, ns, r) &:= \text{if } v = \bot \text{ then } \mathit{resolve}(s, \mathit{prefix}(ns), r) \text{ else } \langle ns, v\rangle
\end{aligned}$$

where $v = \mathit{find}(s, ns \oplus r)$ ■

[3] A namespace is a reference that refers to the depth of a tree of stores where the operation should be performed.

Example.

$$\begin{aligned} s &= \langle \mathtt{a}, \langle \mathtt{b}, \varnothing_{\mathcal{S}} \rangle :: \varnothing_{\mathcal{S}} \rangle :: \varnothing_{\mathcal{S}} \\ \mathit{resolve}(s, \varnothing_{\mathbb{I}}, \mathtt{a} :: \varnothing_{\mathbb{I}}) &= \langle \varnothing_{\mathbb{I}}, \mathtt{a} :: \mathtt{b} :: \varnothing_{\mathbb{I}}, \mathtt{a} :: \varnothing_{\mathbb{I}} \rangle \\ \mathit{resolve}(s, \mathtt{a} :: \varnothing_{\mathbb{I}}, \mathtt{a} :: \varnothing_{\mathbb{I}}) &= \langle \varnothing_{\mathbb{I}}, \mathtt{a} :: \mathtt{b} :: \varnothing_{\mathbb{I}}, \mathtt{a} :: \varnothing_{\mathbb{I}} \rangle \\ \mathit{resolve}(s, \mathtt{a} :: \mathtt{b} :: \varnothing_{\mathbb{I}}, \mathtt{c} :: \varnothing_{\mathbb{I}}) &= \langle \varnothing_{\mathbb{I}}, \bot \rangle \end{aligned}$$

Definition 3.20 (*resolvelink*).
The *resolvelink* function looks a link reference in given store within given namespace. If the resolution value is another link reference, then it must be resolved first until it finds a non link reference value. Since there might be a cyclical ($\boldsymbol{err}_4$), then every visited link reference is kept in an accumulator. Before resolution, the function will check whether the link reference is already in the accumulator and then produces $\boldsymbol{err}_4$ if such situation exists.

$\mathit{resolvelink} : \mathcal{S} \times \mathcal{R} \times \mathcal{R} \times \mathcal{L} \to \mathcal{V}_\bot$
$\mathit{resolvelink}(s, ns, r, \langle \mathit{link}, r_l \rangle) := \mathit{getlink}(s, ns, r, r_l, \{\})$

$\mathit{getlink} : \mathcal{S} \times \mathcal{R} \times \mathcal{R} \times \mathcal{R} \times \mathcal{P}(\mathcal{R}) \to \mathcal{V}_\bot$
$\mathit{getlink}(s, ns, r, r_l, \mathit{acc}) :=$
  if $r_l \in \mathit{acc}$ then $\boldsymbol{err}_4$
  else if $v_p = \langle \mathit{link}, r_m \rangle$ then $\mathit{getlink}(s, \mathit{prefix}(ns_q), r, r_m, \mathit{acc} \cup \{r_l\})$
  else if $ns_q \subseteq_{\mathcal{R}} r$ then $\boldsymbol{err}_5$
  else $\langle ns_q, v_p \rangle$

where: $\langle ns_p, v_p \rangle = \mathit{resolve}(s, ns, r_l)$, and $ns_q = ns_p \oplus r_l$ ■

The following property ensures that function *resolvelink* will always detect any cyclic link reference.

Proposition 3.21. *If function resolvelink is used to resolve a cyclic link reference, then it will produce an error.*

Proof. See appendix A.2 □

Proposition 3.21 shows that *resolvelink* can detect the (*explicit*) cyclic link reference. Another type of cyclical that may exist is what we call as *implicit* cyclic link reference. Consider the specification in figure 3.4. Attribute `comp2` has a link reference that refers to its parent. This can cause a non-terminate valuation because the link reference will be copied during prototype expansion, and then whenever this new link

```
1 // the production compiler never terminates
2 sfConfig extends {
3   comp1 extends {
4     comp2 comp1;
5   }
6 }
```

Figure 3.4: An example specification with implicit cyclic link reference.

reference is resolved, then another link reference is copied, and it continues infinitely. Note that the current SF compiler (version 3.0.18) will not terminate when processing this specification. This problem will be addressed later in definition 3.27.

Definition 3.22 (*copy*).
The ***copy*** function copies every attribute from the second store to the first store at the given prefix (*pfx*).

$$copy : \mathcal{S} \times \mathcal{S} \times \mathcal{R} \to \mathcal{S}$$

$$\begin{aligned} copy(s_1, \varnothing_{\mathcal{S}}, pfx) &:= s_1 \\ copy(s_1, \langle id, v\rangle :: s_2, pfx) &:= copy\left(bind\left(s_1, pfx \oplus id, v\right), s_2, pfx\right) \end{aligned}$$

■

Example.

$$copy(\langle \mathtt{a}, 1\rangle :: \varnothing_{\mathcal{S}}, \langle \mathtt{b}, 2\rangle :: \varnothing_{\mathcal{S}}, \mathtt{a} :: \varnothing_{\mathbb{I}}) = \textbf{\textit{err}}_1$$

$$copy(\langle \mathtt{a}, \varnothing_{\mathcal{S}}\rangle :: \varnothing_{\mathcal{S}}, \langle \mathtt{b}, 2\rangle :: \varnothing_{\mathcal{S}}, \mathtt{a} :: \varnothing_{\mathbb{I}}) = \langle \mathtt{a}, \langle \mathtt{b}, 2\rangle :: \varnothing_{\mathcal{S}}\rangle :: \varnothing_{\mathcal{S}}$$

$$copy(\langle \mathtt{a}, \langle \mathtt{b}, 2\rangle :: \varnothing_{\mathcal{S}}\rangle :: \varnothing_{\mathcal{S}}, \langle \mathtt{c}, 3\rangle :: \varnothing_{\mathcal{S}}, \mathtt{a} :: \varnothing_{\mathbb{I}}) = \langle \mathtt{a}, \langle \mathtt{b}, 2\rangle :: \langle \mathtt{c}, 3\rangle :: \varnothing_{\mathcal{S}}\rangle :: \varnothing_{\mathcal{S}}$$

$$copy(\langle \mathtt{a}, \langle \mathtt{b}, 2\rangle :: \langle \mathtt{c}, 3\rangle :: \varnothing_{\mathcal{S}}\rangle :: \varnothing_{\mathcal{S}}, \langle \mathtt{c}, 4\rangle :: \varnothing_{\mathcal{S}}, \mathtt{a} :: \varnothing_{\mathbb{I}}) = \langle \mathtt{a}, \langle \mathtt{b}, 2\rangle :: \langle \mathtt{c}, 4\rangle :: \varnothing_{\mathcal{S}}\rangle :: \varnothing_{\mathcal{S}}$$

Definition 3.23 (*inherit*).
The ***inherit*** copies values from a given prototype (referred by *proto*) to the target store (referred by *r*). The prototype may be located in a higher-level namespace, hence the use of *resolve* to locate the corresponding store. If the resolution value is a link reference, then it must be resolved first using function *resolvelink*.

$$inherit : \mathcal{S} \times \mathcal{R} \times \mathcal{R} \times \mathcal{R} \to \mathcal{S}$$

$$\begin{aligned} inherit(s, ns, proto, r) \quad := \quad & \text{if } v_p \in \mathcal{S} \text{ then } copy(s, v_p, r) \\ & \text{else if } v_p \in \mathcal{L} \text{ and } v_p = \langle link, r_q\rangle \text{ then} \\ & \quad \text{if } v_q \in \mathcal{S} \text{ then } copy(s, v_q, r) \text{ else } \textbf{\textit{err}}_7 \\ & \text{else } \textbf{\textit{err}}_6 \end{aligned}$$

where $\langle ns_p, v_p\rangle = resolve(s, ns, proto)$, and $\langle ns_q, v_q\rangle = resolvelink(s, ns, r, \langle link, r_q\rangle)$ ■

Note that $\boldsymbol{err}_6$ occurs if the value of the prototype is not a store.

Example.

$inherit(\langle \mathtt{a}, 1\rangle :: \langle \mathtt{b}, \varnothing_{\mathcal{S}}\rangle :: \varnothing_{\mathcal{S}}, \varnothing_{\mathbb{I}}, \mathtt{a} :: \varnothing_{\mathbb{I}}, \mathtt{b} :: \varnothing_{\mathbb{I}}) = \boldsymbol{err}_6$

$inherit(\langle \mathtt{a}, \langle \mathtt{c}, 3\rangle :: \varnothing_{\mathcal{S}}\rangle :: \langle \mathtt{b}, 2\rangle :: \varnothing_{\mathcal{S}}, \varnothing_{\mathbb{I}}, \mathtt{a} :: \varnothing_{\mathbb{I}}, \mathtt{b} :: \varnothing_{\mathbb{I}}) = \boldsymbol{err}_7$

$inherit(\langle \mathtt{a}, \langle \mathtt{c}, 3\rangle :: \varnothing_{\mathcal{S}}\rangle :: \langle \mathtt{b}, \varnothing_{\mathcal{S}}\rangle :: \varnothing_{\mathcal{S}}, \varnothing_{\mathbb{I}}, \mathtt{a} :: \varnothing_{\mathbb{I}}, \mathtt{b} :: \varnothing_{\mathbb{I}}) =$

$\quad \langle \mathtt{a}, \langle \mathtt{c}, 3\rangle :: \varnothing_{\mathcal{S}}\rangle :: \langle \mathtt{b}, \langle \mathtt{c}, 3\rangle :: \varnothing_{\mathcal{S}}\rangle :: \varnothing_{\mathcal{S}}$

Definition 3.24 (*replacelink*).

The ***replacelink*** function resolves the link reference value and then replaces it with the resolution (non-link reference) value. It invokes the *accept* function to replace all link reference values in all sub-stores.

$replacelink : \mathcal{S} \times \mathcal{R} \times (\mathbb{I} \times \mathcal{V}) \times \mathcal{R} \to \mathcal{S}$

$$
\begin{array}{rcl}
replacelink(s, ns, \langle id, v\rangle, ns_s) & := & \text{if } v \in \mathcal{L} \text{ then} \\
 & & \quad \text{if } v_p = \perp \text{ then } \boldsymbol{err}_8 \\
 & & \quad \text{else if } v_p \in \mathcal{S} \text{ then } accept(s_p, r_p, v_p, ns_p) \\
 & & \quad \text{else } s_p \\
 & & \text{else if } v \in \mathcal{S} \text{ then } accept(s, r_p, v, r_p) \\
 & & \text{else } s
\end{array}
$$

where $r_p = ns \oplus id$, $\langle ns_p, v_p\rangle = resolvelink(s, ns_s, r_p, v)$, and $s_p = bind(s, r_p, v_p)$ ■

Note that $\boldsymbol{err}_8$ occurs whenever the link reference value is invalid i.e. the resolution value is undefined ($\perp$).

Definition 3.25 (*accept*).

The ***accept*** function performs the second pass by visiting every element of a store and then passing the element to the *replacelink* function in order to resolve all link references.

$accept : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{R} \to \mathcal{S}$

$$
\begin{array}{rcl}
accept(s, ns, \varnothing_{\mathcal{S}}, ns_s) & := & s \\
accept(s, ns, c :: s_p, ns_s) & := & accept(s_q, ns, s_p, ns_s)
\end{array}
$$

where $s_q = replacelink(s, ns, c, ns_s)$ ■

3.2.3 Valuation Functions

The valuation functions in this section show how each element of the abstract syntax of an SF specification is evaluated. Evaluation of a complete SF specification yields a store $s \in \mathcal{S}$.

Definition 3.26 (Terminals).

The terminal symbols are evaluated in the obvious way, as described in section 2.3.2, using functions with the following signatures:

$$\begin{array}{rlrl} \mathbf{Bool}: & \mathit{Boolean} \to \mathbb{B} & \mathbf{Num}: & \mathit{Number} \to \mathbb{N} \\ \mathbf{Str}: & \mathit{String} \to \mathbb{S} & \mathbf{I}: & \mathit{Identifier} \to \mathbb{I} \\ \mathbf{Null}: & \mathit{NullValue} \to \mathcal{N} & & \end{array}$$

■

For example:

$$\mathbf{Num}[\![\texttt{42}]\!] := 42 \qquad \mathbf{Bool}[\![\texttt{false}]\!] := \mathit{False}$$

Definition 3.27 (References).

Both types of reference are evaluated to a list of identifiers:

$$\begin{array}{rl} \mathbf{LR}: & \mathit{LinkReference} \to \mathcal{R} \to \mathcal{L} \\ \mathbf{DR}: & \mathit{DataReference} \to \mathcal{R} \\ \mathbf{R}: & \mathit{Reference} \to \mathcal{R} \end{array}$$

$$\begin{array}{rcl} \mathbf{LR}[\![\mathrm{R}]\!] & := & \lambda(r).\ \text{if } r_p \subseteq_{\mathcal{R}} r \text{ then } \boldsymbol{err}_4 \text{ else } \langle \mathit{link}, r_p \rangle \\ \mathbf{DR}[\![\mathrm{R}]\!] & := & \mathbf{R}[\![\mathrm{R}]\!] \\ \mathbf{R}[\![\mathrm{I}_1, ..., \mathrm{I}_n]\!] & := & \mathbf{I}[\![\mathrm{I}_1]\!] :: ... :: \mathbf{I}[\![\mathrm{I}_n]\!] :: \varnothing_{\mathbb{I}} \end{array}$$

where $r_p = \mathbf{R}[\![\mathrm{R}]\!]$ ■

Notice that the **LR** function will produce $\boldsymbol{err}_4$ whenever the link reference (r_p) is the prefix of the variable's reference (r) – this detects the implicit cyclic link reference.

Definition 3.28 (Vectors).

Vectors are evaluated by evaluating each element:

$$\mathbf{Vec}: \mathit{Vector} \to (\mathbb{V})^*$$

$$\begin{array}{rcl} \mathbf{Vec}[\![\mathrm{BV}_1, ..., \mathrm{BV}_n]\!] & := & \mathbf{BV}[\![\mathrm{BV}_1]\!] :: ... :: \mathbf{BV}[\![\mathrm{BV}_n]\!] :: \varnothing_{\mathbb{V}} \\ \mathbf{Vec}[\![\varepsilon]\!] & := & \varnothing_{\mathbb{V}} \end{array}$$

■

Definition 3.29 (Basic values).

A basic value (BV) is one of the basic element types:

$$\mathbf{BV}: \mathit{BasicValue} \to \mathbb{V}$$

$$\begin{array}{rclrcl} \mathbf{BV}[\![\mathrm{Bool}]\!] & := & \mathbf{Bool}[\![\mathrm{Bool}]\!] & \mathbf{BV}[\![\mathrm{Num}]\!] & := & \mathbf{Num}[\![\mathrm{Num}]\!] \\ \mathbf{BV}[\![\mathrm{Str}]\!] & := & \mathbf{Str}[\![\mathrm{Str}]\!] & \mathbf{BV}[\![\mathrm{DR}]\!] & := & \mathbf{DR}[\![\mathrm{DR}]\!] \\ \mathbf{BV}[\![\mathrm{Vec}]\!] & := & \mathbf{Vec}[\![\mathrm{Vec}]\!] & \mathbf{BV}[\![\mathrm{Null}]\!] & := & \mathbf{Null}[\![\mathrm{Null}]\!] \end{array}$$

■

Definition 3.30 (Prototype).
A prototype is a sequence of blocks or references. Blocks are evaluated directly, while references are first resolved (in the current context) and then evaluated. Composition proceeds right-to-left (since defined values override any corresponding values in an extended prototype).

$$\mathbf{P}: \quad \mathit{Prototype} \to \mathcal{R} \times \mathcal{R} \times \mathcal{S} \to \mathcal{S}$$

$$\begin{aligned}
\mathbf{P}[\![\mathrm{B}\ \mathrm{P}]\!] &:= \lambda(ns,r,s).\ \mathbf{P}[\![\mathrm{P}]\!](ns,r,\mathbf{B}[\![\mathrm{B}]\!](r,s)) \\
\mathbf{P}[\![\mathrm{R}\ \mathrm{P}]\!] &:= \lambda(ns,r,s).\ \mathbf{P}[\![\mathrm{P}]\!](ns,r,\mathit{inherit}(s,ns,\mathbf{R}[\![\mathrm{R}]\!],r)) \\
\mathbf{P}[\![\varepsilon]\!] &:= \lambda(ns,r,s).\ s
\end{aligned}$$

■

Definition 3.31 (Value).
A value is either a basic value, a prototype, or a link reference. Basic values and link references are entered directly in the store. Prototypes are first evaluated.

$$\mathbf{V}: \quad \mathit{Value} \to \mathcal{R} \times \mathcal{R} \times \mathcal{S} \to \mathcal{S}$$

$$\begin{aligned}
\mathbf{V}[\![\mathrm{BV}]\!] &:= \lambda(ns,r,s).\ \mathit{bind}(s,r,\mathbf{BV}[\![\mathrm{BV}]\!]) \\
\mathbf{V}[\![\mathrm{LR}]\!] &:= \lambda(ns,r,s).\ \mathit{bind}(s,r,\mathbf{LR}[\![\mathrm{LR}]\!](r)) \\
\mathbf{V}[\![\mathrm{P}]\!] &:= \lambda(ns,r,s).\ \mathbf{P}[\![\mathrm{P}]\!](ns,r,\mathit{bind}(s,r,\varnothing_{\mathcal{S}}))
\end{aligned}$$

■

Definition 3.32 (Assignment).
To assign a value to a reference, the store entry for the reference is updated to contain the value.

$$\mathbf{A}: \quad \mathit{Assignment} \to \mathcal{R} \times \mathcal{S} \to \mathcal{S}$$

$$\mathbf{A}[\![\mathrm{R}\ \mathrm{V}]\!] := \lambda(ns,s).\ \mathbf{V}[\![\mathrm{V}]\!](ns,\ ns \oplus r,\ s)$$

■

Definition 3.33 (Block).
A block is a sequence of assignments. These are recursively evaluated left-to-right with the store resulting from one assignment being used as input to the next assignment.

$$\mathbf{B}: \quad \mathit{Block} \to \mathcal{R} \times \mathcal{S} \to \mathcal{S}$$

$$\begin{aligned}
\mathbf{B}[\![\mathrm{A}\ \mathrm{B}]\!] &:= \lambda(ns,s).\ \mathbf{B}[\![\mathrm{B}]\!](ns,\mathbf{A}[\![\mathrm{A}]\!](ns,s)) \\
\mathbf{B}[\![\varepsilon]\!] &:= \lambda(ns,s).\ s
\end{aligned}$$

■

Definition 3.34 (SF Specification).
In the first pass, a complete SF Specification is obtained by evaluating a block, in

the context of an empty store $\varnothing_{\mathcal{S}}$ and a reference $\varnothing_{\mathbb{I}}$ to the root namespace, without resolving any link reference. In the second pass, all link references were resolved and the evaluation of the main `sfConfig` component is returned (other components are ignored - see figure 3.1).

$\mathbf{SF}: \quad \mathit{SFSpecification} \rightarrow \mathcal{S}$

Let $r = \texttt{sfConfig} :: \varnothing_{\mathbb{I}}$, $s_1 = \mathbf{B}[\![\,\mathrm{B}\,]\!](\varnothing_{\mathbb{I}}, \varnothing_{\mathcal{S}})$, and $v_1 = \mathit{find}(s_1, r)$

$\mathbf{SF}[\![\,\mathrm{B}\,]\!] \quad := \quad$ if $v_2 \in \mathcal{S}$ then v_2 else $\boldsymbol{err}_{10}$

where $s_2 =$ if $v_1 \in \mathcal{S}$ then $\mathit{accept}(s_1, r, v_1, r)$ else $\boldsymbol{err}_{10}$

and $v_2 = \mathit{find}(s_2, r)$ ■

Note that s_1 holds the result store of the first pass, while s_2 holds the result store of the second pass. It is an error ($\boldsymbol{err}_{10}$) if the main `sfConfig` element is not a store (for example, if it is a basic value).

There are several facts about this SF semantics. First, notice that that there is no type-checking in binding the value to the variable (see definition 3.31). Every variable can be assigned with any value because SF is a dynamic-typed language.

Second, every prototype reference is resolved using function *resolve* (it is indirectly through *inherit* and *copy* – see definition 3.23 and 3.22). Whenever the prototype reference is not exist at the current-level store, then it will be searched at the upper-level one, and continue until it reaches the top-level.

Third, the second pass evaluation resolves every link reference using function *resolvelink* (through function *accept* and *replacelink*), which detects and produces an error ($\boldsymbol{err}_4$) whenever a cyclic link reference exists.

The last key fact is that the existence of the dereference value of every *data reference* is not checked, which is similar with the behaviour of the current SF production compiler.

3.2.4 Correctness

We define the correctness properties that should be held by the semantics. The first one is that the valuation functions should always terminate given a finite input of SF specification. The second one is that the store and its children stores, as the product of the valuation process, should have unique identifiers. And the third one is that for every reference whose dereference value is not equal undefined, then its prefix has a dereference value of a store. These three properties are formally defined in the following theorems.

Theorem 3.35. *Assume specification SF ∈ SFSpecification and SF is finite, then the valuation of* $\boldsymbol{SF}[\![SF]\!]$ *is always terminate.*

The proof of the above theorem requires us to show that every valuation function is always terminate, given a finite SF specification. Currently, we are unable to provide a complete proof. However, we provide a sketch of proof which gives some essential parts of the proof. This could be guidance to develop a complete proof.

Sketch of Proof. Since the specification is finite and every data reference is not resolved by the valuation functions, then there are two possible cases that can cause a non-terminate valuation i.e. a cyclic link reference and a cyclic prototype.

Because function *resolvelink* is used to resolve every link reference and proposition 3.21 holds, then every cyclic link reference produces an error which terminates the valuation. Another case is the *implicit* cyclic link reference i.e. whenever an attribute is assigned with a link reference that refers to its parent component. Assume r_v is the variable's reference that will be assigned by a link reference, and r_c is the reference of the variable's parent component, then $r_c \subseteq_{\mathcal{R}} r_v$ (see example[4]). Based on the condition branch in *LinkReference*, an error will be produced which terminates the valuation.

Because every prototype is directly resolved and expanded, then a cyclical may occur only whenever the prototype reference is referring to a link reference value. As specified in *Prototype*, every prototype expansion is done using function *inherit*. Based on definition 3.23, *inherit* uses *resolvelink* to resolve any link reference. Since proposition 3.21 holds then any cyclic reference will produce an error which terminates the valuation.

Since the valuation always terminates whenever there is a cyclic link reference or a cyclic prototype, then the statement holds. □

Theorem 3.36. *Assume specification SF ∈ SFSpecification, if* $\boldsymbol{SF}[\![SF]\!] = s$, $s \in \mathcal{S}$ *then s has unique identifiers and* $\forall s_i \subset_{\mathcal{S}} s : s_i$ *has unique identifiers.*

Sketch of Proof. The proof should show that the statement holds for every valuation function in definition §3.2.3. Fortunately, we can focus on the valuation functions that are binding a value to a store i.e. *Value* and *Prototype*, while others can be ignored since there is no binding process in their definitions. Thus, the proof should only show that the statement holds for valuation function *Prototype* and *Value*.

[4]Assume we have store $s = \langle \mathtt{a}, \langle \mathtt{c}, 3\rangle :: \varnothing_{\mathcal{S}}\rangle :: \langle \mathtt{b}, \varnothing_{\mathcal{S}}\rangle :: \varnothing_{\mathcal{S}}$, then $r_c = \mathtt{a} :: \varnothing_{\mathbb{I}}$ is the reference of component a, $r_v = \mathtt{a} :: \mathtt{c} :: \varnothing_{\mathbb{I}}$ is the reference of attribute c of component a, and $r_c \subseteq_{\mathcal{R}} r_v$.

As defined in definition 3.2.3, the first equation of *Prototype* has a binding process where store s passed by function *Value*, and then it is passed to function *inherit* – *inherit* uses function *copy*, and *copy* uses *bind* to perform the binding (see definition 3.23 and 3.22). Since the proof is basing that the statement holds by every valuation function, then s is a store that has unique identifiers. Because s is passed to *bind* and proposition 3.18 holds, then it is valid to say that the first equation always returns a store with unique identifiers. The last two equations of *Prototype* only return a store from other valuation function. Since the proof is basing that the statement holds for every valuation function, then it is valid to say that the statement holds for the last two equations. Thus, the statement holds for *Prototype*.

As defined in definition 3.2.3, every equation of *Value* uses function *bind* to bind a value to store s which is passed by *Assignment*. Since the proof is basing that the statement holds by every valuation function, then s is a store that has unique identifiers. Because proposition 3.18 holds, then it is valid to say that the *Value* always returns a store with unique identifiers.

Since the statement holds for valuation function *Prototype* and *Value*, then the statement holds. □

Theorem 3.37. *Assume specification SF* $\in$ *SFSpecification, if* $\mathcal{SF}[\![SF]\!]$, $s \in \mathcal{S}$ *then* $\forall r \in \mathcal{R} \,.\, find(s,r) \neq \bot \Rightarrow find(s, prefix(r)) \in \mathcal{S}$.

Proof. Since proposition 3.16 holds then the statement holds. □

3.2.5 Discussion

3.2.5.1 Store

The choice of a list structure to represent the store perhaps was not the best option. One of the implications of this choice is that the proof of theorem 3.36 is required to show that the uniqueness property of identifiers in the store is maintained by the semantic functions. Representing the store in terms of sets, or more abstractly as a function $\mathcal{S} : \mathbb{I} \to \mathcal{V}_\bot$, would have simplified the semantics.

However, the formalisation of the semantics of SmartFrog language was largely motivated by the need for a new implementation of the compiler whose common features are compatible with the current production compiler, while supporting additional

```
1 sfConfig extends {
2   a b;
3   b 1;
4   c a;
5 }
```

(a) Forward link reference.

```
1 sfConfig extends {
2   a:b extends {
3     c 2;
4   }
5   a extends {
6     b 1;
7   }
8 }
```

(b) Forward placement.

Figure 3.5: Examples of forward references.

features related to the planning of configuration changes[5]. Therefore, we chose a rather concrete representation that enables us to translate the semantics fairly directly to a functional programming language (Scala and OCaml). This will give us confidence that the compiler is correct.

On the other hand, the experiment results show that the current production compiler (version 3.0.18) is actually preserving the position of every element in the store whenever its value is replaced[6]– this is also confirmed by the language author. This makes our choice of a list structure to represent a store is correct, in particular for simulating the actual behaviour of the production compiler. One of the advantages of using a list is that the order of the elements are deterministic. This can then be exploited for some purposes, for example: we can simply use **diff** to directly compare the output of our compilers and the production compiler for validation.

3.2.5.2 Forward References

The production compiler supports two types of "forward references": any variable can be used before it is defined either on the right hand side of an assignment (a *forward link reference* is shown in figure 3.5a) or on the left hand side (a *forward placement* is shown in figure 3.5b). On the other hand, the semantics only supports one type which is the *forward link reference* while the *forward placement* is illegal.

Supporting forward link references increases the complexity of the semantics since an additional pass (see definition 3.34) must be performed in order to find and resolve every existing link reference. It is interesting that the development of this feature in the semantics identified an issue with the production compiler which fails to termi-

[5]These features are available in SFP language that is described in the last half of this chapter.

[6]This means that the order of the variables, at the time they are declared, must be the same to the order of the variables in the compilation output, regardless of any modification to their values.

```
1 sfConfig extends {
2   a extends {
3     b extends { c 1; }
4   }
5   a:b:c 3;
6   a 1;
7 }
```

Figure 3.6: Ambiguous forward placement.

nate on the specifications of the form shown in figure 3.4. For this example, link reference `comp1` (line 4) will be copied every time the compiler replaces it with the resolution value which is its parent. After resolution, the same link reference must be resolved over and over again by the compiler infinitely. The semantics prevents this non-termination by prohibiting a link reference to referring to its parent as specified in valuation function **LR** (see §3.2). Since our compilers implemented the semantics, then they can detect such situation and produce an error when the same specifications were given as the input. This proves that the semantics can address this issue. Note that we have discussed this issue with the SmartFrog developer from HP Labs who implemented the production compiler, and it will be addressed in the next version.

There is no loss of functionality in prohibiting the forward placement since the source code can always be re-ordered to avoid it. In addition, the text of required sub-specifications could be included (using `#include`) before it is being used by the assignment. However, since the parts of specification could be created independently by different people, then some extra works may be required to re-ordere the assignments. This is obviously not an easy task in particular when the requirements are defined in many files with multilevel inclusion. Thus, the forward placement is supported by the production compiler.

Supporting forward placement is more difficult than forward link reference. The current production compiler uses three passes to perform this. Clearly, this will make the semantics more complex and harder to be implemented than the current version. This also requires invalidating certain expressions such as shown in figure 3.6 which are valid in the current semantics. In our discussion with Patrick Goldsack, the SF creator, there is a consideration on deprecating the forward placement in the next version. Besides complexity, this is also another reason for us to not extend the semantics to support forward placement.

The production compiler also supports another type of forward reference called as

```
1 sfConfig extends {
2   a extends {
3     b 1;
4   }
5   c extends a
6 }
```

(a) Standard prototype.

```
1 sfConfig extends {
2   c extends a
3   a extends {
4     b 1;
5   }
6 }
```

(b) Forward prototype.

Figure 3.7: Examples of standard and forward prototypes.

"forward prototype reference": any component can be used as the prototype of another component before it is defined. Figure 3.7 shows the difference between standard and forward prototype. Supporting the forward prototype requires another pass by the production compiler which could make the semantics more complex. Note that mutual prototype reference is illegal.

Technically, the source codes with forward prototype references can be re-ordered to avoid any error in using our compilers. During experiments, we only found one specification file[7] in SmartFrog software distribution that contains the forward prototype reference. Since it is illegal in our compilers, we then addressed this problem simply by re-ordering the inclusion file statements so that the required sub-specifications are included before it is being used by others. This re-orderings would not affect the compilation output of the production compiler because it supports the specification with and without the forward prototype reference.

However, the re-ordering could be not a trivial task in particular when the specifications are defined in multi-files with multilevel inclusion. Thus, we add this feature as part of our future works.

3.2.6 SF for Planning

The SF language was originally designed to describe a static configuration state of a system. Although it has the notion of a runtime, but it provides limited features: every component only has fixed states (undeployed, deployed, and running) and fixed state-transitions (deploy, undeploy, start, terminate). In addition, the ordering relationship between the states of components must be defined implicitly in the orders of declaration statements. Thus, it is difficult to use the SF specification for planning of state changes of the components since there is no explicit description about the conditions

[7]The file is: `org/smartfrog/sfcore/workflow/combinators/components.sf`.

of before and after the changes. These descriptions are required for sound reasoning why the state of a component should change and what are the conditions that must be satisfied before the change.

On the other hand, the dynamic-typing of SF allows any variable to be assigned with any value. Any planning process will take into account all possible values that can be assigned to every variable. This dynamic-typing implies that the state search space of planning will be very large – if there are in total m variables and n values, then the search space will be m^n.

The following sections describe an extension of SF in order to address the above issues. First, the notation of action is introduced so that we can explicitly describe the preconditions and effects of every change. This also allows the components to having arbitrary states and transitions. Second, the notation of global constraints is introduced to naturally describe conditions that should be maintained at every state of the system. Third, the static type system is introduced to allow us to explicitly define the domain of every variable. In practice, this can significantly reduce the state search space of planning.

3.3 SFP by Example

SmartFrog for Planning (SFP) language is built based on the SF core subset. They are different in several aspects: SFP is a static typed language while SF is dynamic-typed; SFP has notations to declaratively describe the dynamic aspect of configuration changes i.e. the *global constraints* and the *actions*, while these notations are not available in SF.

SFP type system serves two objectives. First is providing particular safety at compile-time. For example, whenever a variable is assigned with a reference value then the type system ensures that the dereference value exists and its type is compatible with the variable's type. Second is ensuring that a domain is defined for every variable. The smaller the domain, the smaller search-space that the planner will have in the process to find a solution-plan (planning time) – type *server* is preferable than type *machine* or even type *object*, for example. Thus, we can expect that the plan can be found in a shorter time.

On the other side, in some declarative configuration tools such as SmartFrog and LCFG, the workflow of configuration changes are hidden in the implementation of the

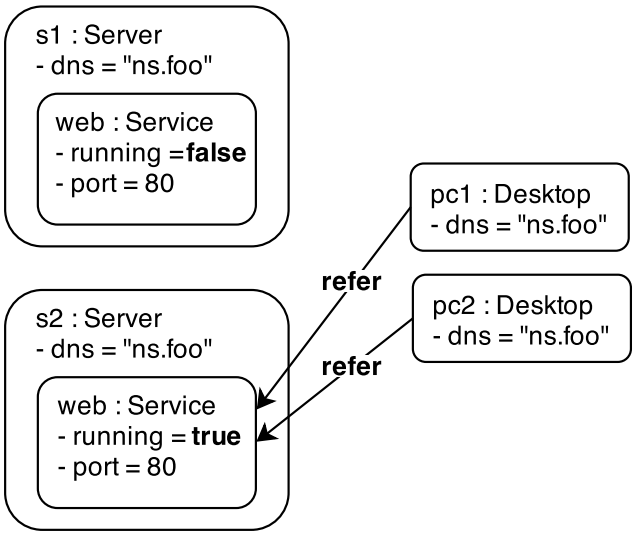


Figure 3.8: The new desired state of the example system in figure 3.1. The new values are using **bold** fonts.

resource component[8]. Thus, if we want to enforce particular workflow which is not implemented in the resource component, then we have to: change the implementation of the resource component, or perform the workflow manually by presenting step-by-step of every intermediate and goal states to the tool. These solutions are clearly time consuming, error prone, and not suitable for unattended use.

One approach to this problem has been the use of manual workflow tools such as provided in Chef, Ansible, SaltStack, and IBM Tivoli Provisioning Manager. However, this still requires that the workflows are computed manually. Even in a small system, a very large number of workflows can be required to cater for every eventuality. And choosing an appropriate workflow to suit the goal state is not always obvious.

SFP provides an alternative approach that is providing the notations of the global constraints and the actions so that the ordering constraints can be expressed as state constraints. Thus, we can add any necessary constraint into the specification where the tool automatically computes the suitable workflow that matches the current state to achieve the desired state. The following example gives a brief introduction on this idea.

Assume that we have a system as described in §3.1 depicted in figure 3.1. We aim to change the configuration to the goal state shown in figure 3.8 where the service in server s_1 is stopped, the service in server s_2 is running, and the desktop clients pc_1

[8]It is the software component that is responsible to implement configuration changes to achieve the desired state.

```
1  // file : model.sfp
2  schema Machine {
3    dns = "ns.foo";
4  }
5  schema Client extends Machine {
6    refer: *Service = null;
7    def redirect(s: Service) {
8      condition { }
9      effect {
10       this.refer = s;
11     }
12   }
13 }
14 schema Service {
15   running = true;
16   port = 80;
17   def start {
18     condition {
19       this.running = false;
20     }
21     effect {
22       this.running = true;
23     }
24   }
25   def stop {
26     condition {
27       this.running = true;
28     }
29     effect {
30       this.running = false;
31     }
32   }
33 }
```

Figure 3.9: SFP specification of the resource model of the system depicted in figure 3.1. It is kept in file **model.sfp**.

and pc_2 are referring to the service in s_2. In addition, we want to maintain particular constraint during the changes i.e. pc_1 and pc_2 always refer to a running service.

To model such configuration, first we need to model each abstract resource in a schema as specified in figure 3.9. Lines 1-3 describe a schema of machine that has an attribute `dns` with default value of a string. Lines 4-14 present a schema of desktop client that extends schema `Machine`. It has an additional attribute `refer` whose type is a reference of service. Besides attributes, this schema has an action defined in lines 6-13. The action has a parameter `s` whose type is a reference of service, a condition[9]

[9]Precondition before execution.

```
1  include "system1-model.sfp";
2  main {
3    s1 isa Machine {
4      web isa Service { }
5    }
6    s2 extends s1, {
7      web.running = false;
8    }
9    pc1 isa Client {
10     refer = s1.web;
11   }
12   pc2 pc1;
13 }
```

Figure 3.10: SFP specification of the current state of the system depicted in figure 3.1.

```
1  include "system1-model.sfp";
2  main {
3    s1 isa Machine {
4      web isa Service {
5        running = false;
6      }
7    }
8    s2 extends s1, {
9      web.running = true;
10   }
11   pc1 isa Client {
12     refer = s2.web;
13   }
14   pc2 pc1;
15   global {
16     pc1.refer.running = true;
17     pc2.refer.running = true;
18   }
19 }
```

Figure 3.11: SFP specification of the desired state of the system depicted figure 3.8.

(lines 7-9), and an effect[10] (lines 10-12) – keyword `this` refers to the parent object. Lines 15-30 define a schema of service that has two attributes, each has a type and a default value. The schema has two actions (lines 18-23 and lines 24-29) that can start or stop the service. Note that SFP treats schema as static, which means that the structure of a schema cannot be modified outside its declaration.

Figure 3.10 specifies the current state of the system. In practice the tool automatically generates this current state by aggregating the state of every agent. On the other hand, the desired state is specified in figure 3.11. Although this specification looks

[10]Postcondition after execution.

similar to the SF specifiction in figure 3.1, but it is different in several things. First, keyword `sfConfig` is replaced with `main` only to differentiating SFP with SF. Second, each variable has a type which can be defined explicitly or inferred automatically from its value. For instance, variable `pc1.refer` has a type of reference of service and the compiler ensures that the dereference value does exist[11]. Third, every component that implements particular schema will inherit both its attributes and actions. For example, since service `s1.web` implements schema `Service` then it inherits attributes `running` and `port` as well as actions `start` and `stop`. Finally, the specification has global constraints defined in lines 15-18. These constraints ensure that pc_1 and pc_2 always refer to a running service.

3.4 Formal Semantic of SFP Language

The SFP formal semantic consists of five parts: the core abstract syntax (§3.4.1), the core type system (§3.4.2), the core valuation functions (§3.4.3), the global constraint valuation functions (§3.4.4), and the action valuation functions (§3.4.5).

3.4.1 Core Abstract Syntax

Definition 3.38 (SFP Core Terminal Symbols).
These are the basic symbols of the language inherited from SF that appear in the source code:

$$\begin{aligned} \text{Bool} &\in \textit{Boolean} \\ \text{Num} &\in \textit{Number} \\ \text{Str} &\in \textit{String} \\ \text{Null} &\in \textit{NullValue} \\ \text{I} &\in \textit{Identifier} \end{aligned}$$

■

Definition 3.39 (SFP Core Non Terminal Symbols).
These are the non-terminal elements of the syntax directly inherited from SF:

[11] The compiler produces an error if the dereference value is not exist or the type is not compatible.

$$
\begin{aligned}
\mathrm{B} &\in \textit{Block} \\
\mathrm{A} &\in \textit{Assignment} \\
\mathrm{P} &\in \textit{Prototype} \\
\mathrm{R} &\in \textit{Reference} \\
\mathrm{DR} &\in \textit{DataReference} \\
\mathrm{LR} &\in \textit{LinkReference} \\
\mathrm{Vec} &\in \textit{Vector} \\
\mathrm{BV} &\in \textit{BasicValue}
\end{aligned}
$$

These are the non-terminal elements of the syntax inherited from SF with modifications:

$$\mathrm{V} \in \textit{Value}$$

And these are the new non-terminal elements of the syntax (not available in SF):

$$
\begin{aligned}
\mathrm{SFP} &\in \textit{SFPSpecification} \\
\mathrm{SC} &\in \textit{SFPContext} \\
\mathrm{S} &\in \textit{Schema} \\
\mathrm{SS} &\in \textit{SuperSchema} \\
\mathrm{G} &\in \textit{GlobalConstraint} \\
\mathrm{Ac} &\in \textit{Action} \\
\mathrm{TV} &\in \textit{TypeVar}
\end{aligned}
$$

■

Definition 3.40 (Type Syntax).
A basic type is either a boolean, a number, a string, an object, or any identifier used as the name of a schema. A type is either a basic type, a vector of basic type, a reference of basic type, a null, an action, or a global constraint.

$$
\begin{aligned}
\mathrm{T} &::= \tau \mid \texttt{[]}\,\tau \mid \texttt{*}\,\tau \mid \texttt{null} \mid \texttt{act} \mid \texttt{glob} \\
\tau &::= \texttt{bool} \mid \texttt{num} \mid \texttt{str} \mid \texttt{obj} \mid \mathrm{I}
\end{aligned}
$$

■

Definition 3.41 (Core Abstract Syntax). The non-terminals are defined by the following *abstract syntax*. Note that the syntax do not specify the details of the concrete syntax (see appendix B.1 for the SFP concrete syntax). In addition, the syntax of the global constraint and the action are described in separate subsections.

$$
\begin{aligned}
\text{SFP} &::= \text{SC} \\
\text{SC} &::= \text{A SC} \mid \text{S SC} \mid \text{G SC} \mid \varepsilon \\
\text{B} &::= \text{A B} \mid \text{G B} \mid \varepsilon \\
\text{A} &::= \text{R TV V} \\
\text{V} &::= \text{BV} \mid \text{LR} \mid \text{SS P} \mid \text{Ac} \\
\text{P} &::= \text{R P} \mid \text{B P} \mid \varepsilon \\
\text{BV} &::= \text{Bool} \mid \text{Num} \mid \text{Str} \mid \text{Null} \mid \text{Vec} \mid \text{DR} \\
\text{LR} &::= \text{R} \\
\text{DR} &::= \text{R} \\
\text{Vec} &::= (\text{BV})^*
\end{aligned}
$$

$$
\begin{aligned}
\text{R} &::= \text{I R} \mid \text{I} \\
\text{S} &::= \text{I SS B} \\
\text{SS} &::= \text{I} \mid \varepsilon \\
\text{TV} &::= \text{T} \mid \varepsilon
\end{aligned}
$$

■

3.4.2 Type System

This section introduces a static type system for SFP language. The type system serves two objectives. The first is providing particular safety at compile-time. For example, a variable is assigned a reference value where the type system ensures that the dereference value exists and its type is compatible with the variable's type. The second is providing that a domain is defined for every variable. The smaller the domain, the smaller search-space that the planner will have in the process to find a solution-plan (planning time). Thus, we can expect that the plan can be found in a shorter time.

The static type system is formally defined as a *proof system* that is the set of rules determining the assignment of type to particular expression and applied recursively, each of which is in the form of:

$$
\text{(Name)} \quad \frac{\textit{premises}}{\textit{conclusion}}
$$

where premises are separated by a space or written on multiple lines. Each rule requires an environment to resolve any variable name or reference value. Every variable can be represented by its absolute reference in the context of root store, that is: $r \in \mathcal{R}$.

Consider the following example:

```
1 main {
2   machine {
3     dns = "foo.com";
4   }
5 }
```

The above specification has three variables represented by references: `main`, `main.machine`, and `main.machine.dns`.

Definition 3.42 (Type environment).

$\Gamma ::= r_1 : \tau_1, ..., r_n : \tau_n$ type environment, $r \in \mathcal{R}$

$dom(\Gamma) = \{r_1, ..., r_n\}$ environment domain

■

A type environment is a map of variables' reference to their type. Note that the form of $r_1 : \tau_1, ..., r_n : \tau_n$ means that type τ_i is assigned to variable r_i.

The formal specification starts with the type system judgements that may be made with respect to a typing environment Γ. The type system always starts with an empty environment represented by symbol $\varnothing$.

Definition 3.43 (Typing Judgements).

$\Gamma \vdash \diamond$ Γ is a well-formed environment

$\vdash \tau$ τ is a well-formed type

$\tau <: \tau'$ type τ is a sub-type of τ'

$\Gamma \vdash r : \tau$ in Γ, variable r has type τ

■

There are three judgements: an environment is well-formed; a type is a well-formed type; and a type is a subtype of another.

Definition 3.44 (Rules of well-formed environments and types).

(Env) $\dfrac{}{\varnothing \vdash \diamond}$ (Env Var) $\dfrac{\Gamma \vdash \diamond \quad \vdash \tau \quad r \notin dom(\Gamma)}{\Gamma, r : \tau \vdash \diamond}$ (Type Bool) $\dfrac{}{\vdash \texttt{bool}}$ (Type Num) $\dfrac{}{\vdash \texttt{num}}$ (Type Str) $\dfrac{}{\vdash \texttt{str}}$

(Type Null) $\dfrac{}{\vdash \texttt{null}}$ (Type Object) $\dfrac{}{\vdash \texttt{obj}}$ (Type Action) $\dfrac{}{\vdash \texttt{act}}$ (Type Global) $\dfrac{}{\vdash \texttt{glob}}$ (Type Vec) $\dfrac{\vdash \tau}{\vdash []\tau}$

(Type Ref) $\dfrac{\vdash \tau}{\vdash *\tau}$ (Type Schema) $\dfrac{id \;\; id_s \;\; \mathrm{B} \in \mathrm{S} \quad id_s \neq \varepsilon \Rightarrow \vdash id_s}{\vdash id}$

where id, id_s are schema identifiers. ■

A well-formed environment is either empty (Env) or a map of variable references to types (Env Var). A well-formed type is either a boolean, a number, a string, an object, a null, a vector, a reference, a schema, an action, or a global constraint. Every type has corresponding vector (Type Vec) and reference types (Type Ref). A schema declaration introduces new type of its identifier, but if it has a super schema then the super schema must be a well-formed type (Type Schema). Notice that these rules and the following ones are using the abstract syntax in §3.4.1.

Definition 3.45 (Rules of subtyping).

(Schema Subtype) $$\frac{s\ s'\ \mathrm{B} \in \mathrm{S} \quad s' \neq \varepsilon}{s <: s'}$$

(Object Subtype) $$\frac{s\ \varepsilon\ \mathrm{B} \in \mathrm{S}}{s <: \mathtt{obj}}$$

(Reflex) $$\frac{\vdash \tau}{\tau <: \tau}$$

(Trans) $$\frac{\tau <: \tau' \quad \tau' <: \tau''}{\tau <: \tau''}$$

(Vec Subtype) $$\frac{\tau <: \tau'}{[]\tau <: []\tau'}$$

(Ref Subtype) $$\frac{\tau <: \tau'}{*\tau <: *\tau'}$$

(Ref Null) $$\frac{\vdash *\tau}{\mathtt{null} <: *\tau}$$

■

A schema is a subtype of its super schema (Schema Subtype), or it is a subtype of object type if it does not have a super schema (Object Subtype). A type is reflexive (Reflex), and it is also transitive (Trans). The subtyping property of a type is also applied to its type vector (Vec Subtype) and reference (Ref Subtype). A null is a subtype of all type reference (Ref Null).

Definition 3.46 (Rules of type assignment).

(Subsum) $$\frac{\Gamma \vdash r : \tau \quad \Gamma \vdash \tau <: \tau'}{\Gamma \vdash r : \tau'}$$

(Var Res) $$\frac{\Gamma \vdash \diamond \quad (r : \tau) \in \Gamma}{\Gamma \vdash r : \tau}$$

(Vec) $$\frac{\Gamma \vdash v_i : \tau \quad \forall i \in 1..n}{\Gamma \vdash [v_1, ..., v_n] : []\tau}$$

(Bool) $$\frac{\Gamma \vdash \diamond \quad v \in \mathrm{Bool}}{\Gamma \vdash v : \mathtt{bool}}$$

(Num) $$\frac{\Gamma \vdash \diamond \quad v \in \mathrm{Num}}{\Gamma \vdash v : \mathtt{num}}$$

(Str) $$\frac{\Gamma \vdash \diamond \quad v \in \mathrm{Str}}{\Gamma \vdash v : \mathtt{str}}$$

(Null) $$\frac{\Gamma \vdash \diamond \quad v \in \mathrm{Null}}{\Gamma \vdash v : \mathtt{null}}$$

(DR) $$\frac{v \in \mathrm{DR} \quad \Gamma \vdash v : \tau}{\Gamma \vdash v : *\tau}$$

(LR) $$\frac{v \in \mathrm{LR} \quad \Gamma \vdash v : \tau}{\Gamma \vdash v : \tau}$$

(Act) $$\frac{\Gamma \vdash \diamond \quad v \in \mathrm{Ac}}{\Gamma \vdash v : \mathtt{act}}$$

(Glob) $$\frac{\Gamma \vdash \diamond \quad v \in \mathrm{G}}{\Gamma \vdash v : \mathtt{glob}}$$

(Proto1) $$\frac{\varepsilon\ \varepsilon \in \mathrm{V}}{\Gamma \vdash \varepsilon\ \varepsilon : \mathtt{obj}}$$

(Proto2) $$\frac{\varepsilon\ p_1 ... p_n \in \mathrm{V} \quad p_1 \in \mathrm{B}}{\Gamma \vdash \varepsilon\ p_1 ... p_n : \mathtt{obj}}$$

(Proto3) $$\frac{\varepsilon\ p_1 ... p_n \in \mathrm{V} \quad p_1 \in \mathrm{R} \quad \Gamma \vdash p_1 : \tau_1 \quad (p_i \in \mathrm{R} \wedge \Gamma \vdash p_i : \tau_i) \Rightarrow \tau_i <: \tau_1 \quad \forall i \in 2..n}{\Gamma \vdash \varepsilon\ p_1 ... p_n : \tau_1}$$

(Proto4)
$$\frac{\tau\ p_1 ... p_n \in \mathrm{V} \quad (p_i \in R \wedge \Gamma \vdash p_i : \tau_i) \Rightarrow \tau_i <: \tau \quad \forall i \in 1..n}{\Gamma \vdash \tau\ p_1 ... p_n : \tau}$$

(Proto5)
$$\frac{\tau\ \varepsilon \in V \quad \Gamma \vdash \tau}{\Gamma \vdash \tau\ \varepsilon : \tau}$$

(Assign1)
$$\frac{r\ \varepsilon\ v \in \mathrm{A} \quad r \notin dom(\Gamma) \quad \Gamma \vdash v : \tau}{\Gamma, r : \tau \vdash \diamond}$$

(Assign2)
$$\frac{r\ \varepsilon\ v \in \mathrm{A} \quad \Gamma \vdash r : \tau \quad \Gamma \vdash v : \tau' \quad \tau' <: \tau}{\Gamma \vdash \diamond}$$

(Assign3)
$$\frac{r\ \tau\ v \in \mathrm{A} \quad r \notin dom(\Gamma) \quad \Gamma \vdash v : \tau' \quad \tau' <: \tau}{\Gamma, r : \tau \vdash \diamond}$$

(Assign4)
$$\frac{r\ \tau'\ v \in \mathrm{A} \quad \Gamma \vdash r : \tau \quad \Gamma \vdash v : \tau'' \quad \tau'' <: \tau' <: \tau}{\Gamma \vdash \diamond}$$

■

The subsumption rule (Subsum) means that a variable with a type can be assigned with any value of its supertypes. The next rule (Var Res) is the type resolution of a variable. The rule (Vec) specifies that every element of a vector should have the same type, and the element's type becomes the vector's type. This is followed by four rules (Bool, Num, Str, Null) that specify the type of four basic values: a boolean, a number, a string, and a null. Rule (LR) states that the type of a link reference is the same as the source type. However, rule (DR) states that the type of the data reference is a reference of the source type. Note that there is no reference of reference. Thus, if the source type is already a reference, then the type of the data reference is the same as the source type. Rules (Act) and (Glob) define that the actions and the global constraints have specific types – these will be used later by the planner to extract the actions and the global constraints from the specification.

The rule (Proto1) defines that the type of an empty prototype value, without any schema, is a plain object. In the rule (Proto2), if there is no schema and the first prototype is a block then the prototype value is a plain object. However, in the rule (Proto3), if the first prototype is a reference prototype then other reference prototypes must be a subtype of the first, and the type of the first reference prototype is the type of the value. But, whenever a schema preceeds the prototypes (Proto4), then all reference prototypes must be a subtype of the schema, and the schema is the type of the value. (Proto5) defines a schema and without any prototype.

The last four rules are for assigning a value to a variable. In the rule (Assign1), if the variable has not been defined and there is no explicit type then the type of value is used as the variable's type. However, in (Assign2), if the variable is defined then the

value's type must be a subtype of the variable's type. In (Assign3), the value's type must be a subtype of an explicit type which becomes the variable's type if it has not been defined. Finally, the rule (Assign4) states that an explicit type and the value's type must be subtypes of a defined variable's type.

3.4.3 Core Valuation Functions

This section presents value-level valuation functions of SFP core syntax in definition §3.41. Definition of TV (*TypeVar*) is omitted since it is only used by the type system. Since the semantics are reusing definition 3.26 (Bool, Num, Str, I, Null), 3.27 (R, DR, LR), 3.28 (Vec), 3.29 (BV), and 3.30 (P), then their definitions will be omitted as well. Other symbols are evaluated using the following functions.

Definition 3.47 (**Super Schema**)**.**
A super schema is first resolved and then evaluated to copy its attributes to the target object or subschema.

$$\mathbf{SS}: \quad \mathit{SuperSchema} \to \mathcal{R} \times \mathcal{S} \to \mathcal{S}$$

$$\begin{aligned} \mathbf{SS}[\![\mathrm{I}]\!] &:= \lambda(r,s)\,.\,\mathit{inherit}(s, \varnothing_{\mathbb{I}}, r_s, r) \\ \mathbf{SS}[\![\varepsilon]\!] &:= \lambda(r,s)\,.\,s \end{aligned}$$

where $r_s = \mathbf{I}[\![\mathrm{I}]\!] :: \varnothing_{\mathbb{I}}$ ■

Definition 3.48 (**Schema**)**.**
A schema is a super schema and a block. The block is evaluated directly, while the super schema is first resolved and then evaluated.

$$\mathbf{S}: \quad \mathit{Schema} \to \mathcal{S} \to \mathcal{S}$$

$$\mathbf{S}[\![\mathrm{I\ SS\ B}]\!] := \lambda s\,.\,\mathbf{B}[\![\mathrm{B}]\!](r, s_v)$$

where $r = \mathbf{I}[\![\mathrm{I}]\!] :: \varnothing_{\mathbb{I}}$, and $s_v = \mathbf{SS}[\![\mathrm{SS}]\!](r, \mathit{bind}(s, r, \varnothing_{\mathcal{S}}))$ ■

Definition 3.49 (**Value**)**.**
A value is either a basic value, a schema-prototype, a link reference, or an action. Basic values, link references, and actions are entered directly in the store. A schema is first evaluated, and then prototypes are evaluated.

$$\mathbf{V}: \quad \mathit{Value} \to \mathcal{R} \times \mathcal{R} \times \mathcal{S} \to \mathcal{S}$$

$$\begin{aligned} \mathbf{V}[\![\mathrm{BV}]\!] &:= \lambda(ns,r,s)\,.\,\mathit{bind}(s, r, \mathbf{BV}[\![\mathrm{BV}]\!]) \\ \mathbf{V}[\![\mathrm{LR}]\!] &:= \lambda(ns,r,s)\,.\,\mathit{bind}(s, r, \mathbf{LR}[\![\mathrm{LR}]\!](r)) \\ \mathbf{V}[\![\mathrm{P\ SS}]\!] &:= \lambda(ns,r,s)\,.\,\mathbf{P}[\![\mathrm{P}]\!](ns, r, s_v) \\ \mathbf{V}[\![\mathrm{Ac}]\!] &:= \lambda(ns,r,s)\,.\,\mathit{bind}(s, r, \mathbf{Ac}[\![\mathrm{Ac}]\!]) \end{aligned}$$

where $s_v = \mathbf{SS}[\![\mathrm{SS}]\!](r, bind(s, r, \varnothing_{\mathcal{S}}))$ ■

Notice that definitions of $\mathbf{V}[\![\mathrm{BV}]\!]$ and $\mathbf{V}[\![\mathrm{LR}]\!]$ are the same with the first and second functions of definition §3.31.

Definition 3.50 (Assignment).
To assign a value to a reference, the store entry for the reference is updated to contain the value.

$$\mathbf{A}: \quad Assignment \to \mathcal{R} \times \mathcal{S} \to \mathcal{S}$$
$$\mathbf{A}[\![\mathrm{R\ TV\ V}]\!] \quad := \quad \lambda(ns, s).\, \mathbf{V}[\![\mathrm{V}]\!](ns, ns \oplus r, s)$$

■

Notice that TV is not evaluated since it is only used by the type system.

Definition 3.51 (SFP Context).
A context is a sequence of attributes, schemata, or global constraints. Attributes, schemata, and global constraints are first evaluated.

$$\mathbf{SC}: \quad SFPContext \to \mathcal{R} \times \mathcal{S} \to \mathcal{S}$$
$$\begin{aligned}
\mathbf{SC}[\![\mathrm{A\ SC}]\!] &:= \lambda(ns, s).\, \mathbf{SC}[\![\mathrm{SC}]\!](ns, \mathbf{A}[\![\mathrm{A}]\!](ns, s)) \\
\mathbf{SC}[\![\mathrm{S\ SC}]\!] &:= \lambda(ns, s).\, \mathbf{SC}[\![\mathrm{SC}]\!](ns, \mathbf{S}[\![\mathrm{S}]\!](s)) \\
\mathbf{SC}[\![\mathrm{G\ SC}]\!] &:= \lambda(ns, s).\, \mathbf{SC}[\![\mathrm{SC}]\!](ns, \mathbf{G}[\![\mathrm{G}]\!](s)) \\
\mathbf{SC}[\![\varepsilon]\!] &:= \lambda(ns, s).\, s
\end{aligned}$$

■

Definition 3.52 (Block).
A block is either a sequence of assignments or global constraints. Assignments are recursively evaluated left-to-right with the store resulting from one assignment being used as input to the next assignment. While global constraints are evaluated as a conjunction formula.

$$\mathbf{B}: \quad Block \to \mathcal{R} \times \mathcal{S} \to \mathcal{S}$$
$$\begin{aligned}
\mathbf{B}[\![\mathrm{A\ B}]\!] &:= \lambda(ns, s).\, \mathbf{B}[\![\mathrm{B}]\!](ns, \mathbf{A}[\![\mathrm{A}]\!](ns, s)) \\
\mathbf{B}[\![\mathrm{G\ B}]\!] &:= \lambda(ns, s).\, \mathbf{B}[\![\mathrm{B}]\!](ns, \mathbf{G}[\![\mathrm{G}]\!](s)) \\
\mathbf{B}[\![\varepsilon]\!] &:= \lambda(ns, s).\, s
\end{aligned}$$

■

Definition 3.53 (SFP Specification).
At the first pass, a complete SFP Specification is obtained by evaluating a sequence

of SFP contexts, in the context of an empty store $\varnothing_{\mathcal{S}}$ and a reference $\varnothing_{\mathbb{I}}$ to the root namespace, without resolving any link reference. At the second pass, all link references were resolved and the evaluation of the main `main` component is returned (other components are ignored).

$\mathbf{SFP}: \quad \mathit{SFPSpecification} \rightarrow \mathcal{S}$

Let $r = \texttt{main} :: \varnothing_{\mathbb{I}}$, $s_1 = \mathbf{SC}[\![\mathrm{SC}]\!](\varnothing_{\mathbb{I}}, \varnothing_{\mathcal{S}})$, and $v_1 = \mathit{find}(s_1, r)$

$$
\begin{array}{lll}
\mathbf{SFP}[\![\mathrm{SC}]\!] & := & \text{if } v_2 \in \mathcal{S} \text{ then} \\
 & & \quad | \quad \text{if } v_g = \bot \text{ then } v_2 \\
 & & \quad | \quad \text{else if } v_g \in \mathbb{C} \text{ then } \mathit{bind}(v_2, r_g, v_g) \\
 & & \quad | \quad \text{else } \boldsymbol{err}_{12} \\
 & & \text{else } \boldsymbol{err}_{11}
\end{array}
$$

where $s_2 = \text{if } v_1 \in \mathcal{S} \text{ then } \mathit{accept}(s_1, r, v_1, r) \text{ else } \boldsymbol{err}_{11}$

$v_2 = \mathit{find}(s_2, r)$, $r_g = \texttt{global} :: \varnothing_{\mathbb{I}}$, $v_g = \mathit{find}(s_1, r_g)$ ■

Note that s_1 holds the result of the first pass, while s_2 holds the result of the second pass. It is an error ($\boldsymbol{err}_{11}$) if the main element (`main`) is not a store (for example, if it is a basic value). Since the global constraints is at the top-level store (referred by $\texttt{global} :: \varnothing_{\mathbb{I}}$), then it should be reassigned to the main element.

Definition of *GlobalConstraint* (**G**) and *Action* (**Ac**) will be given in following subsections.

3.4.4 Global Constraint

The global constraints are logic formulas that must be true at any state of the system. This section describes how to construct the functions that represent the logic formulas of the constraints as defined in the specification. Since the constraints can be defined in separate declarations, then the final global constraint is a conjunction of these declarations. For example, consider the following:

```
1  main {
2    x = 1;
3    y {
4      z = true;
5      global {
6        x in [1, 2, 3];
7      }
8    }
9    global {
10     y = true;
11   }
12 }
```

The above specification is equivalent to:

```
main {
  x = 1;
  y {
    z = true;
  }
  global {
    x in [1, 2, 3];
    y = true;
  }
}
```

Any reference in the constraint can be *prevail*, *nested* (reference to another reference), or *invalid*.

Definition 3.54 (Reference Classification).

Assume $\sigma \in \mathcal{S}$ and $r \in \mathcal{R}$, then:

r is a *prevail* reference of σ

iff $\mathit{find}(\sigma, r) \neq \perp$, or

r is a *nested* reference of σ

iff $\mathit{find}(\sigma, r) = \perp \wedge \exists r_s.(r_s \neq \varnothing_{\mathbb{I}} \wedge r_s \subset_{\mathcal{R}} r) \Rightarrow \mathit{find}(\sigma, r_s) \neq \perp$, or

r is an *invalid* reference of σ

iff r is not a *prevail* or *nested* reference. ■

For example, consider the following specification:

```
schema DBService {
  running = true;  // type: bool
}
schema WebService {
  running = true;          // type: bool
  db : *DBService = null;  // type: *DBService
}
schema Client {
  refer : WebService = null;
}
main {
  s1 isa DBService;  // type: DBService
  s2 isa WebService { // type: WebService
    db = s1;   // type: *DBService
  }
  pc isa Client {  // type: Client
    refer = s2;  // type: *WebService
  }
  global {
    pc.refer = web;              // valid statement
    pc.refer.running = true;     // valid statement
    pc.refer.db.running = true;  // valid statement
  }
}
```

In above, lines 19-23 define the global constraints: the first is a *prevail* and the second and third are *nested* references. The prevail reference's type can be easily inferred from the type environment using the reference. However, this could not be used to determine the type of the nested reference.

We can use the type information available in the schema to determine the type of a nested reference. For example, `pc.refer.running` is the reference to be inferred. First, we must determine the longest prevail prefix of `pc.refer.running` which is available in the type environment. In this case, `main.pc.refer` is the longest prevail with type `WebService`. Afterwards, we find an attribute of `WebService` which is equal to the first identifier of the non prevail prefix i.e. `running`. The combination of the schema's identifier and the non prevail prefix (`WebService.running`) is then used to get the type of the original reference i.e. boolean. Similar steps are also used to determine the type of the second nested reference (line 22) which will return a boolean type as well.

Unfortunately, we have not been able to formally define the type checking rules for the global constraints. We leave this as part of the future works.

Definition 3.55 (**Non Terminal Symbols**)**.**
These are the non-terminal elements of basic constraint syntax:

$$\begin{aligned} \text{And} &\in \textit{Conjunction} \\ \text{Or} &\in \textit{Disjunction} \\ \text{Eq} &\in \textit{Equal} \\ \text{Ne} &\in \textit{NotEqual} \\ \text{Im} &\in \textit{Implication} \\ \text{Not} &\in \textit{Negation} \\ \text{ML} &\in \textit{MemberOfList} \end{aligned}$$

These are the non-terminal elements of the global constraints syntax:

$$\begin{aligned} \text{G} &\in \textit{GlobalConstraint} \\ \text{CS} &\in \textit{ConstraintStatement} \end{aligned}$$

■

Definition 3.56 (**Abstract Syntax**)**.**
The non-terminals are defined by the following *abstract syntax*:

$$
\begin{array}{rcl}
\text{G} & ::= & \text{And} \\
\text{And} & ::= & (\text{CS})^* \\
\text{Or} & ::= & (\text{CS})^* \\
\text{CS} & ::= & \text{Eq} \mid \text{Ne} \mid \text{Not} \mid \text{Im} \mid \text{And} \mid \text{Or} \mid \text{ML} \\
\text{Eq} & ::= & \text{R BV} \\
\text{Ne} & ::= & \text{R BV} \\
\text{Im} & ::= & \text{And And} \\
\text{Not} & ::= & \text{CS} \\
\text{ML} & ::= & \text{R Vec}
\end{array}
$$

Note that symbols R, BV, and Vec are the same as the symbols of the core syntax (see definition 3.39).

■

First, we define the constraint domain that represents the valuation result of the constraint.

Definition 3.57 (Constraint Domain). $\mathbb{C} = \mathcal{S} \rightarrow \mathbb{B}$, is the constraint domain which is a function that receives a store as parameter, and returns *True* if the store satisfies the constraints, otherwise it returns *False*. ■

Definition 3.58 (Basic Constraint).
Every basic constraint is evaluated to form a corresponding function of logic formula.

$\mathbf{And} : \mathit{Conjunction} \rightarrow \mathbb{C}$

$$
\begin{array}{rcl}
\mathbf{And}[\![\text{CS}_1, ..., \text{CS}_n]\!] & := & \lambda s.\, \mathbf{CS}[\![\text{CS}_1]\!](s) \wedge ... \wedge \mathbf{CS}[\![\text{CS}_n]\!](s) \\
\mathbf{And}[\![\varepsilon]\!] & := & \lambda s.\, \mathit{True}
\end{array}
$$

$\mathbf{Or} : \mathit{Disjunction} \rightarrow \mathbb{C}$

$$
\begin{array}{rcl}
\mathbf{Or}[\![\text{CS}_1, ..., \text{CS}_n]\!] & := & \lambda s.\, \mathbf{CS}[\![\text{CS}_1]\!](s) \vee ... \wedge \mathbf{CS}[\![\text{CS}_n]\!](s) \\
\mathbf{Or}[\![\varepsilon]\!] & := & \lambda s.\, \mathit{True}
\end{array}
$$

$\mathbf{Eq} : \mathit{Equal} \rightarrow \mathbb{C}$

$$\mathbf{Eq}[\![\text{R BV}]\!] := \lambda s.\, \mathit{find}(s, \mathbf{R}[\![\text{R}]\!]) = \mathbf{BV}[\![\text{BV}]\!]$$

$\mathbf{Ne} : \mathit{NotEqual} \rightarrow \mathbb{C}$

$$\mathbf{Ne}[\![\text{R BV}]\!] := \lambda s.\, \neg(\mathit{find}(s, \mathbf{R}[\![\text{R}]\!]) = \mathbf{BV}[\![\text{BV}]\!])$$

$\mathbf{Im} : \mathit{Implication} \rightarrow \mathbb{C}$

$$\mathbf{Im}[\![\text{And}_1\ \text{And}_2]\!] := \lambda s.\, \mathbf{And}[\![\text{And}_1]\!](s) \Rightarrow \mathbf{And}[\![\text{And}_2]\!](s)$$

$\mathbf{Not} : \mathit{Negation} \rightarrow \mathbb{C}$

$$\mathbf{Not}[\![\text{CS}]\!] := \lambda s.\, \neg\, \mathbf{CS}[\![\text{CS}]\!](s)$$

$$\mathbf{ML} : \mathit{MemberOfList} \to \mathbb{C}$$
$$\mathbf{ML}[\![\mathrm{R\ \ Vec}]\!] \quad := \quad \lambda s\,.\,\mathit{find}(s, \mathbf{R}[\![\mathrm{R}]\!]) \in \mathbf{Vec}[\![\mathrm{Vec}]\!]$$

■

Definition 3.59 (Constraint Statement).

A constraint statement is either of the basic constraint.

$$\mathbf{CS} : \mathit{ConstraintStatement} \to \mathbb{C}$$

$$\begin{array}{rcl@{\qquad}rcl} \mathbf{CS}[\![\mathrm{Eq}]\!] & := & \mathbf{Eq}[\![\mathrm{Eq}]\!] & \mathbf{CS}[\![\mathrm{Ne}]\!] & := & \mathbf{Ne}[\![\mathrm{Ne}]\!] \\ \mathbf{CS}[\![\mathrm{Not}]\!] & := & \mathbf{Not}[\![\mathrm{Not}]\!] & \mathbf{CS}[\![\mathrm{IL}]\!] & := & \mathbf{IL}[\![\mathrm{IL}]\!] \\ \mathbf{CS}[\![\mathrm{Im}]\!] & := & \mathbf{Im}[\![\mathrm{Im}]\!] & \mathbf{CS}[\![\mathrm{FE}]\!] & := & \mathbf{FE}[\![\mathrm{FE}]\!] \\ \mathbf{CS}[\![\mathrm{And}]\!] & := & \mathbf{And}[\![\mathrm{And}]\!] & \mathbf{CS}[\![\mathrm{Or}]\!] & := & \mathbf{Or}[\![\mathrm{Or}]\!] \end{array}$$

■

Definition 3.60 (Global Constraint).

The final global constraint is a conjunction of all global constraint declarations.

$$\mathrm{G} : \mathit{GlobalConstraint} \to \mathcal{S} \to \mathcal{S}$$
$$\begin{array}{rcll} \mathbf{G}[\![\mathrm{And}]\!] & := & \lambda s. & \text{if } g_s = \bot \text{ then } \mathit{bind}(s, r, g_c) \\ & & & \text{else if } g_s \in \mathbb{C} \text{ then } \mathit{bind}(s, r, f) \\ & & & \text{else } \boldsymbol{err}_{12} \end{array}$$

where $r = \texttt{global} :: \varnothing_{\mathbb{I}}$, $g_s = \mathit{find}(s, r)$, $g_c = \mathbf{And}[\![\mathrm{And}]\!]$, and $f = \lambda s'\,.\,g_s(s') \wedge g_c(s')$

■

Notice that since the global constraint must be satisfied at every state, then the expression of *MemberOfList* can be used to reduce the size of the domain of a particular variable by defining a list of value that the variable can have. Assume we have the following:

```
1 main {
2   x = 1;  // type: num
3   global {
4     x in [1, 2, 3, 4, 5];  // defined by person A
5   }
6   global {
7     x in [0, 1, 2];  // defined by person B
8   }
9 }
```

Line 2 shows that x's domain is $\mathbb{R}$. However, line 4 specifies that x's value is always either 1, 2, 3, 4, or 5. Thus, its domain can be redefined as $\mathbb{R} \cap \{1,2,3,4,5\} = \{1,2,3,4,5\}$. Line 5 reduces the domain further more where the final domain of x is $\{1,2,3,4,5\} \cap \{0,1,2\} = \{1,2\}$. This can help the administrators to precisely define the domain of the variable, and it can reduce the planning search space.

3.4.5 Action

The introduction of the action notations in SFP was largely motivated by the need for allowing the administrator to declaratively describe the state changes of configurations, which then can be exploited by an automated planner to automatically generate a workflow between any two viable states. We adopt the PDDL-style for describing an action where each action has a name, parameters, preconditions, effects, and a cost.

Definition 3.61 (**Non Terminal Symbols**)**.**
These are the non-terminal symbols of action syntax:

$$\begin{aligned} \text{Ac} &\in \textit{Action} \\ \text{Pa} &\in \textit{Parameters} \\ \text{Co} &\in \textit{Cost} \\ \text{Cd} &\in \textit{Condition} \\ \text{Ef} &\in \textit{Effect} \end{aligned}$$

Definition 3.62 (**Abstract Syntax**)**.**
The non-terminals are defined by the following *abstract syntax*:

$$\begin{aligned} \text{Ac} &::= \text{Pa Co Cd Ef} \\ \text{Pa} &::= (\text{I T})^* \\ \text{Co} &::= \text{Num} \mid \varepsilon \\ \text{Cd} &::= \text{And} \mid \varepsilon \\ \text{Ef} &::= (\text{R BV})^+ \end{aligned}$$

Note that symbols I, T, Num, R and BV are the same as the symbols of SFP core (see §3.4.1). While symbols And is the same as the symbol of SFP global constraints (see definition 3.55). ■

Definition 3.63 (**Action Domain**)**.**
Action domain is a 4-tuple $\mathbb{A} = (\mathbb{I})^* \times \mathbb{R} \times \mathbb{C} \times (\mathbb{E})^*$ whose elements are a list of parameter, an action cost, the preconditions, and the effects respectively, where:
$\mathbb{E} = \{e_i | e_i : \mathcal{S} \to \mathcal{S}\}$ ■

Definition 3.64 (**Effect**)**.**
An effect is a list of variable assignment (order does not matter) when the action is executed. The left-hand side is the reference of the variable, while the right-hand side is the value to be assigned. Every reference must be prevail.

Ef : $\textit{Effect} \to \mathbb{E}$

$$\mathbf{Ef}[\![\mathrm{R}_1\ \mathrm{BV}_1,...,\mathrm{R}_n\ \mathrm{BV}_n]\!] \quad := \quad \lambda s\,.\, \mathit{eff}_i(...(\mathit{eff}_1(s)))$$

where $\mathit{eff}_i = \lambda s\,.\, \mathit{bind}(s, \mathbf{R}[\![\mathrm{R}]\!]_i, \mathbf{BV}[\![\mathrm{BV}]\!]_i)$ ■

Definition 3.65 (Precondition).

The preconditions is a constraint that must be satisfied before executing the action.

$\mathbf{Cd} : \mathit{Condition} \to \mathbb{C}$

$$\begin{aligned} \mathbf{Cd}[\![\mathrm{And}]\!] &:= \mathbf{And}[\![\mathrm{And}]\!] \\ \mathbf{Cd}[\![\varepsilon]\!] &:= \lambda s\,.\, \mathit{True} \end{aligned}$$

■

Definition 3.66 (Cost).

A cost represents the action's preference value comparing to the others. This can be used by the planner to find a global optimal solution by searching a plan whose total cost of actions is the minimum.

$\mathrm{Co} : \mathit{Cost} \to \mathbb{R}$

$$\begin{aligned} \mathbf{Co}[\![\mathrm{Num}]\!] &:= \mathbf{Num}[\![\mathrm{Num}]\!] \\ \mathbf{Co}[\![\varepsilon]\!] &:= 1 \end{aligned}$$

■

Definition 3.67 (Parameter).

A parameter is a list of free variables that can be used within the action context. Each free variable has a type which will be used to determine the substitution value for generating the grounded-actions.

$\mathrm{Pa} : \mathit{Parameter} \to (\mathbb{I})*$

$$\begin{aligned} \mathbf{Pa}[\![\mathrm{I}\ \mathrm{T}\ \mathrm{Pa}]\!] &:= \langle \mathbf{I}[\![\mathrm{I}]\!] \rangle :: \mathbf{Pa}[\![\mathrm{Pa}]\!] \\ \mathbf{Pa}[\![\varepsilon]\!] &:= \varnothing \end{aligned}$$

■

Definition 3.68 (Action).

The parameter, the cost, the precondition, and the effect are first evaluated.

$\mathbf{Ac} : \mathit{Action} \to \mathbb{A}$

$$\mathbf{Ac}[\![\mathrm{Pa}\ \mathrm{Co}\ \mathrm{Cd}\ \mathrm{Ef}]\!] \quad := \quad \langle\, \mathbf{Pa}[\![\mathrm{Pa}]\!],\, \mathbf{Co}[\![\mathrm{Co}]\!],\, \mathbf{Cd}[\![\mathrm{Cd}]\!],\, \mathbf{Ef}[\![\mathrm{Ef}]\!] \,\rangle$$

■

3.4.6 Discussion

3.4.6.1 Action

Notice that the action's effect can only assign a variable with a basic value: a boolean, a number, a string, a vector, a null, or a reference. It cannot assign (and create) a new

object or delete an existing one. The main reason for this design is because every configuration task will be compiled into a classical planning problem where the number of objects must be finite.

On the other hand, every reference in the action effect must be a prevail reference. Any nested reference will make the effect to be non-deterministic because it requires particular conditions (e.g. a particular variable should not be *null*) to be satisfied before applying the assignment.

In the classical planning, every action must not have a parameter – this is called as a grounded-action. However, since SFP allows an action to have parameters, then a grounding-process is required in order to transform actions with parameters to grounded-actions (without parameters) by substituting the parameters with appropriate values.

One of motivations of the introduction of static typing in SFP was the need for reducing the number of possible values that can substitute a parameter of action. This is based on the fact that any value can substitute an untyped-parameter. On the other hand, only values with particular type can substitute a typed-parameter. Thus, a static typing helps the grounding algorithm to producing less number of grounded-actions which implies small search space, since a planning problem is actually a combinatorial problem of actions. However, we have not been able to formally define the type-checking rules for the actions. We leave this as part of the future works.

3.4.6.2 Loose Specification

Currently, the SFP language only supports a "rigid" specification where every variable only has one possible value at the desired state. Some previous works, such as [Hewson et al., 2012], introduced a way to describe a "loose" specification where the desired state is described as constraints. A similar idea can be easily adopted in SFP.

For example, we can introduce a notation to express *any* value, which can be assigned to a particular variable. This special value means that the variable can have any value at the final state as long as the value is a member of the variable's domain.

Another way is that a new type of constraint, called *final*, is introduced in order to allow us to define the constraints which are only applied at the final state of the system. We can use constraints or SAT solver in order to generate the possible final states that satisfy the *final* constraints. Since there will be more than one goal state that can be achived by the workflow, then a particular strategy must be implemented during planning. §4.2.6 describes the details of this strategy.

3.5 Summary

As summary, this chapter has presented the formal semantics of the core subset of SmartFrog (SF) language. Some useful properties, such as termination, have been provided and proved. The formal semantics have been used as a precise and independent reference for developing alternative SF compilers[12]. In addition, the development of the semantics helped us to identify an issue on the current SF production compiler which fails to terminate on the specification of a particular form. The solution of this issue has been provided in the formal semantics.

On the other hand, the chapter has presented the SFP language which extends SF with notations of global constraints and actions. Unlike SF, SFP is a static-typed language. The semantics of SFP has been formalised. However, the type checking rules for the global constraints and the actions have not been formalised – they are part of future works. This SFP formal semantics has been used as an independent reference for developing the SFP compiler.

[12] At the time this thesis was written, there are three SF compilers which were developed only based on the formal semantics.

Chapter 4

Planning Configuration Changes

Every classical planning problem uses a restricted planning model where the goal of the problem only specifies the final state that should be achieved by the plan. However, in practice, we might want to specify the *extended goal* of the problem which concerns not only on the final state, but also on every visited state during execution. This extended goal is commonly found in the problems of the system configuration domain – the configuration specification defines the desired state of the system as well as the *global constraints* that must be preserved during configuration changes.

[Herry et al., 2011] has shown that an off-the-shelf classical planner can be used to automatically generate a plan as the solution of a configuration problem. However, this approach requires the global constraints to be directly encoded as preconditions associated with some actions. Thus, a change to the constraint forces us to modify the actions. In real situations this is impractical: the specification of the actions are commonly written by a software engineer or expert who has a deep knowledge of the software which the administrator does not have. Determining whether an action must be modified or not may be as hard as the planning itself, since the constraints for execution could require arbitrary states to be achieved by previous actions. In addition, a modification may not be allowed due to a lack of permission or a license violation, for example.

The automated planning community has identified and addressed this restriction issue in the Planning Domain Definition Language version 3 (PDDL3). PDDL3 extends the previous version (PDDL2[1]) by allowing us to express the extended goal as the state trajectory constraints using a set of *modal operators*. One of the modal operators is `always`, which is used to express the global constraints.

[1]The gola of PDDL2 only specifies the final state.

This chapter presents two contributions described in two different sections. The first section presents a domain independent technique that compiles a planning problem with extended goals into a classical planning problem. The compilation result can then be solved by any classical planner. The second section presents a technique to translate a configuration task defined in SFP language into a classical planning problem. The translation result is then given as input to an off-the-shelf classical planner in order to generate the solution plan which can be executed to bring the system from the current to the desired configuration state while preserving the global constraints.

4.1 Planning with Extended Goal

In system configuration, the notion of the global constraints means the constraints that must be satisfied at all states visited by the plan. Based on the definition 2.2 of FDR task, we can formally define a planning problem with global constraint as the following.

Definition 4.1 (Planning Problem with Global Constraint).
A planning problem with global constraint is a 5-tuple $\Theta = \langle V, A, s_0, s_g, s_\beta \rangle$ where:

- $V = \{v_1, ..., v_n\}$ is a set of state variables, each of which is associated with a finite domain D_v. If $d \in D_v$ then $v = d$ is an atom. A partial variable assignment over V is a function s on some subset of V such that $s(v) \in D_v$, wherever $s(v)$ is defined. If $s(v)$ is defined for all $v \in V$, then s is a state;

- A is a set of actions, each of which is a 4-tuple $\langle name, cost, pre, eff \rangle$, where *name* is a unique symbol, $cost \in \mathbb{R}^{0+}$ is a non-negative cost, while *pre* and *eff* are partial variable assignments called preconditions and effects;

- s_0 is a state called an initial state, s_β is a partial variable assignment called the global constraints and s_g is a partial variable assignment called the goal. If π is the solution plan of Θ and $\langle s_0, s_1, ..., s_n \rangle$ are the states visited by π, then the semantics of s_g and s_β are:
  $\langle s_0, s_1, ..., s_n \rangle \models s_g$ iff $s_n \models s_g$
  $\langle s_0, s_1, ..., s_n \rangle \models s_\beta$ iff $\forall i : 0 \leq i \leq n \,.\, s_i \models s_\beta$

■

Based on the above definition, we can solve the planning problem with global constraint by compiling it into an FDR task (as defined in §2.2.3) as the input for

the classical planner to find the solution plan. The main idea of the compilation is enforcing the planner to verify whether the state after executing an action satisfies the global constraint. Note that this compilation technique assumes that the solution plan to the planning problem is a sequential plan.

The compilation can be summarised as follows:

1. An artificial binary state variable v_β is introduced as the flag of the global constraint. At the initial state $v_\beta = \mathit{False}$ which means that the initial state is not satisfying the global constraint. While at the goal state $v_\beta = \mathit{True}$ which means that the goal should satisfy the global constraint;

2. Each original action is assumed to violate the global constraint after its execution by adding atom $v_\beta = \mathit{False}$ into its effect. In addition, each action cannot be executed if the previous state does not satisfy the constraint by adding atom $v_\beta = \mathit{True}$ into its precondition;

3. An artificial action a_β is introduced whose cost is 0, precondition is $s_\beta \wedge (v_\beta = \mathit{False})$ and effect is $v_\beta = \mathit{True}$. During planning, a_β will be immediately selected by the planner after another action in order to ensure that the resulting state satisfies the global constraint.

4.1.1 First-Order Formula

If the global constraint is a first-order formula, then it must be converted into a *Disjunctive Normal Form* (DNF) formula due to the restriction of FDR where each action's precondition must be a conjunction of atoms. If the DNF after conversion is in the form of $\phi_1 \vee \phi_2 \vee ... \vee \phi_n$ where ϕ_i is a conjunction of atoms, then an artificial action a^i_β is introduced for each ϕ_i whose precondition is $\phi_i \wedge (v_\beta = \mathit{False})$ and effect is $v_\beta = \mathit{True}$. During planning, any artificial action can be picked by the planner since the global constraint is true if any conjunction clause of DNF is true.

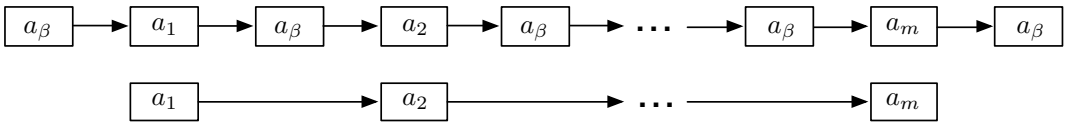


Figure 4.1: The solution plan before (top) and after (bottom) postprocessing.

If the solution plan is found then its length is $2m+1$ where m is the plan's length for the original problem and $m+1$ is the number of artificial actions. Thus, a postprocessing step must be performed to remove the artificial action to get the solution plan for the original problem. Figure 4.1 illustrates a plan before and after the postprocessing.

Intuitively, the compilation can be done in another way: adding the global constraint into the preconditions of each original action. It seems that the planning problem after this alternative compilation is simpler because the artificial variable and action (v_β and a_β) are not being required. However, this could lead to a potential larger number of actions if the global constraint is a first-order formula. This is because the formula must be converted to DNF, where for each DNF conjunctive clause, each original action must be copied and the clause is added into its preconditions to ensure that the previous state satisfies the constraint. This alternative compilation has one less variable compared to the above compilation. However, if there are n original actions and the DNF of the global constraint has p conjunctive clauses, then the problem produced by this alternative compilation will have $n*p$ actions which is higher than the previous compilation technique that has $n+p$ actions. The lower number of actions, the better performance we could expect from the planner since it has less number of actions to be considered during planning.

On the other hand, the conversion of a first-order formula to DNF can be very expensive and the size of the DNF formula can be exponential compared to the original. Implication formula is one of the factors that cause the exponential explosion. For example, in DNF, a logical formula of the following form has 2^n clauses:

$$(\phi_1 \Rightarrow \psi_1) \wedge (\phi_2 \Rightarrow \psi_2) \wedge ... \wedge (\phi_n \Rightarrow \psi_n)$$

Fortunately, there is another way to compile the global constraint formula if it is a conjunction that has a set of *simple implication* clauses.

Definition 4.2 (Simple Implication).
$\phi \Rightarrow \psi$ is a simple implication iff ϕ and ψ are conjunctions of atoms. ■

Based on experience, simple implication formulas are found in many real use-cases[2]. Thus, a special handling of the simple implication would give a benefit because it could minimize the exponential explosion effect, or even eliminate it if the global constraint is purely a conjunction of simple implication clauses.

[2]In system configuration domain, a dependency between component is defined as a simple implication formula.

ϕ	ψ	$\phi \Rightarrow \psi$
true	true	true
true	false	false
false	true	true
false	false	false

Figure 4.2: Truth table of an implication.

The basic idea of compiling the simple implication is based on the truth table shown in figure 4.2. It shows that $\phi \Rightarrow \psi$ is false iff ϕ is true and ψ is false. Thus, some necessary modifications must be done on each action's preconditions in order to ensure that this case will not occur at the state before and after executing the action.

Definition 4.3 defines the compilation rules that must be applied to each original action in order to ensure that the simple implication constraint is not violated during execution. For each rule, above the line is the premise that must be satisfied before modification and below the line is a modification/step that should be done to the action. Note that the rules only modify the action preconditions, while the effects are unchanged.

Definition 4.3 (Simple Implication Compilation Rules).
Assume $a_i = \langle name_i, cost_i, pre_i, eff_i \rangle$ is the original action and $a'_i = \langle name_i, cost_i, pre'_i, eff'_i \rangle$ is the action after modification. Then the following rules are applied to each original action:

a) $$\frac{(pre_i \models \phi \wedge \neg\psi) \vee (eff_i \models \phi \wedge \neg\psi)}{\textbf{delete } a_i}$$

b) $$\frac{pre_i \not\models \neg\phi \wedge pre_i \models \neg\psi}{pre'_i = pre \wedge \neg\phi}$$

c) $$\frac{pre_i \models \phi \wedge pre_i \not\models \psi}{pre'_i = pre_i \wedge \psi}$$

d) $$\frac{eff_i \models \phi \wedge eff_i \not\models \psi}{pre'_i = pre_i \wedge \psi}$$

e) $$\frac{eff_i \not\models \neg\phi \wedge eff_i \models \neg\psi}{pre'_i = pre_i \wedge \neg\phi}$$

■

Rule (a) deletes the action because its preconditions or effects violates the simple implication. Rules (b) and (c) add necessary condition to the preconditions of the original action so that the state before execution does not violate the simple implication. While rules (d) and (e) add additional effects to ensure that the state after execution does not violate the simple implication.

Algorithm 4.1 shows the complete steps to compile a planning problem with global constraint to a classical planning problem. If there are n variables and m actions in the original problem, the global constraint has p simple implications where $p > 0$, and the DNF formula of the global constraint has q conjunction clauses, then the algorithm's complexity is $O(m * p)$ if $m * p \geq q$ or $O(q)$ if $m * p < q$. The number of variables and actions after compilation are $n + 1$[3]and $m + q$ respectively.

Unfortunately, if the global constraint is a complex formula e.g a conjunction of non-simple implication clauses, then the compilation could be expensive since the for-

[3]After compilation, the problem will have one additional state variable v_β as the flag of the global constraint.

Algorithm 4.1 Compile a planning problem with global constraint to a classical planning problem.

Require: V is the set of variables, A is the set of actions, s_0 is the initial state, s_g is the goal, S_β is the first-order formula of the global constraint.

1: **function** COMPILE-GLOBAL(V, A, s_0, s_g, S_β)
2: **if** S_β is a conjunction formula **then**
3: **for each** simple implication clause $(\phi \Rightarrow \psi) \in S_\beta$ **do**
4: **for each** action $a_i \in A$ **do**
5: apply simple implication rules of $(\phi \Rightarrow \psi)$ to a_i ▷ see def. 4.3
6: remove $(\phi \Rightarrow \psi)$ from S_β
7: $S_{\beta DNF} \leftarrow$ convert S_β to DNF
8: $V \leftarrow V \cup v_\beta$ ▷ $D_{v_\beta} = \{True, False\}$
9: **for each** action $a \in A$, $a = \langle name, pre, eff \rangle$ **do**
10: $pre \leftarrow pre \wedge (v_\beta = True)$
11: $eff \leftarrow eff \wedge (v_\beta = False)$
12: **for each** conjunction clause $\vartheta \in S_{\beta DNF}$ **do**
13: $name' \leftarrow$ unique symbol
14: $pre' \leftarrow \vartheta \wedge (v_\beta = False)$
15: $eff' \leftarrow (v_\beta = True)$
16: $A \leftarrow A \cup \{\langle name', 0, pre', eff' \rangle\}$ ▷ add artificial action with cost 0
17: $s_0 \leftarrow s_0 \wedge (v_\beta = False)$
18: $s_g \leftarrow s_g \wedge (v_\beta = True)$
19: **return** $\langle V, A, s_0, s_g \rangle$

mula must be converted to DNF. Perhaps a solution to this problem is to replace FDR with another representation that allows the preconditions of the action to be represented as a Conjunctive Normal Form (CNF) or even a first-order formula. However, changing the representation may affect the heuristic search technique used by the planner. For example, both *causal-graph* (h^{CG}) [Helmert, 2004] and *context-enhanced-additive* (h^{CEA}) [Helmert and Geffner, 2008] heuristics heavily rely on the causal-graph structure generated from FDR – the causal graph represents relations between state variables based on the preconditions and effects of every FDR action. Replacing FDR with a different representation will make the heuristics to be unusable because the causal graph cannot be generated due to the representation changes – the heuristic might be adapted but it is not an easy task. On the other hand, other heuristics such as FastFoward (h^{FF}) [Hoffmann and Nebel, 2001] or Landmark (h^{LM}) [Richter and Westphal, 2010] do not rely on the causal-graph – the heuristics are calculated based on the *relaxed*[4] version of the planning problem. Thus, it is possible to use FF and LAMA in the representation that allows the action's preconditions in CNF or first-order formula. We leave further investigation of this idea for future work.

4.1.2 Uncompilability Constraint

The above compilation scheme preserves the plan existence, which means that if there exists a sequential plan satisfying the goal and the global constraint of the original planning problem, then there also exists a valid sequential plan for the compiled problem. However, it is possible to define a planning problem with a global constraint whose solution plan is a parallel plan that involves parallel actions, but no sequential plan. For example, the global constraint specifies that a table must be lifted at the same time by two robots. This example can be represented as the following.

[4] A *relaxed* problem of the FDR task allows every variable to hold more than one value at particular state. It means that whenever an action assigns a new value to any variable, then the old value will still be kept. Thus, the variable will hold the old and the new values at the next state.

initial state:	goal:
- robot1-hand = empty	- robot1-hand = desk
- robot2-hand = empty	- robot2-hand = desk
action: robot1-lift-desk	action: robot2-lift-desk
precond: robot1-hand = empty	precond: robot2-hand = empty
effect: robot1-hand = desk	effect: robot2-hand = desk

global constraint:
((robot1-hand = desk) $\wedge$ (robot2-hand = desk)) $\vee$
((robot2-hand = empty) $\wedge$ (robot1-hand = empty))

Clearly, only a parallel plan can solve this problem while any sequential plan would not since the problem requires action **robot1-lift-desk** and **robot2-lift-desk** to be executed in parallel. Our compilation technique will not be able to solve this type of problem since its output is a classical planning problem where any classical planner will only generate a sequential plan.

4.1.3 Partial-Order Plan

After compilation, the classical planner will generate a total-order plan which must be executed sequentially. In practice, the sequential execution could become the bottleneck of the system since only one action can be executed at particular time. Thus, it is necessary to have a partial-order plan which enables parallel execution of the actions.

[Veloso et al., 1990] and [Ambite and Knoblock, 2001] have introduced algorithms to generate a partial-order plan from a total-order plan based on STRIPS formalism [Fikes and Nilsson, 1971]. Algorithm 4.2 is an extended version of these algorithms adapted for FDR[5]plan with additional step to maintain the global constraint.

Function GENERATE-PARTIAL-ORDER is called after the classical planner generates the plan for the compiled version of the planning problem. It returns a partial-order plan for the original problem. In lines 2-3 of the algorithm, the preconditions of every artificial action a^i_β is added to the preconditions of a non-artificial action a_i that immediately follows it. This additional condition will ensure that the state before executing action a_i has satisfied the global constraint. Thus, artificial actions can be ignored at the next steps (lines 4-13). Lines 5-9 generates the causal links i.e. finding the actions that support the preconditions of every action. If a_k has an effect that supports the preconditions of a_i where $k < i$ and there is no such action between them that *threats*

[5]See [Helmert, 2009] for more details about the differences between STRIPS and FDR.

Algorithm 4.2 Generate partial-order plan.

Require: A valid total-order plan $\pi = \langle a_\beta^1, a_1, a_\beta^2, a_2, ..., a_\beta^n, a_n, a_\beta^{n+1} \rangle$ generated after compilation, where a_β^i is the artificial action.

1: **function** GENERATE-PARTIAL-ORDER(π)
2: **for each** action $a_i \in \pi$ **do**
3: $Preconditions(a_i) \leftarrow Preconditions(a_i) \wedge Preconditions(a_\beta^i)$
4: **for** $i \leftarrow n$ down-to 1 **do**
5: **for each** $(v = d) \in Preconditions(a_i)$ **do**
6: **choose** $k < i$ such that:
7: 1. $(v = d) \in Effects(a_k)$, and
8: 2. $\nexists l : k < l < i \,.\, (v = d') \in Effects(a_l) \wedge d \neq d'$
9: add order $a_k \prec a_i$
10: **for each** $(v = d) \in Effects(a_i)$ **do**
11: **for** $j \leftarrow (i-1)$ down-to 1 **do**
12: **if** $(v = d') \in Preconditions(a_j) \wedge d \neq d'$ **then**
13: add order $a_j \prec a_i$
14: **return** $\langle \{a_1, a_2, ..., a_n\}, \prec \rangle$

the preconditions then an ordering constraint $a_k \prec a_i$ is added. This selection is non-deterministic, which means that it is possible that there is a_l such that $l > k$ and a_l supports the preconditions of a_i. Lines 10-13 add an ordering constraint $a_j \prec a_i$ if the effects of a_i threat the preconditions of a_j.

[Bäckström, 1994] has shown that the problem of finding an optimal partial-order plan from a total-order plan is NP-hard under several definitions of optimality. The purpose of the non-deterministic selection in algorithm 4.2 (line 6) is to enable exhaustive exploration of every possible action in order to get an optimal partial-order plan. However, exhaustive exploration could be very expensive due to a large number of possible combinations of the selection, in particular when there is a deadline of the searching process. Thus, the algorithm can be set to perform a greedy selection by changing line 6 into: **max** $k < i$. With this change, the algorithm complexity will be polynomial but there is no guarantee that the partial-order plan is globally optimal – for example, it is possible that there is a partial-order plan whose length is less than the one generated by the algorithm.

4.1.4 State Trajectory Constraint of PDDL3

Planning Definition Domain Language (PDDL) is a domain-independent language used in the International Planning Competition (IPC) to model planning problems. [Gerevini et al., 2009] is introducing the latest version of PDDL i.e. PDDL3. Unlike the previous versions, such as PDDL2, PDDL3 has the notions of state trajectory constraints which assert conditions that must be satisfied by the entire sequence of states visited during the execution of the plan. The constraints are expressed in first-order logic formulas over state predicates using six modal operators i.e. `at-end`, `always`, `sometime`, `sometime-after`, `sometime-before`, and `at-most-once`. Definition 4.4 formally specifies the semantics all modal operators.

Definition 4.4 (**Semantics of State Trajectory Constraint** [Gerevini et al., 2009])**.** Given a domain $\mathcal{D}$, a plan π, an initial state s_0, and the states visited by π are $\langle s_0, s_1, ..., s_n \rangle$. If $\mathcal{G}$ is the formula of the state trajectory constraint, then π is valid if it visits states $\langle s_0, s_1, ..., s_n \rangle$ that satisfies the goal: $\langle s_0, s_1, ..., s_n \rangle \models \mathcal{G}$.

The semantics of each modal operator are:

$\langle s_0, s_1, ..., s_n \rangle \models$ (`at-end` ϕ)
iff $s_n \models \phi$;

$\langle s_0, s_1, ..., s_n \rangle \models$ (`always` ϕ)
iff $\forall i : 0 \leq i \leq n \,.\, s_i \models \phi$;

$\langle s_0, s_1, ..., s_n \rangle \models$ (`sometime` ϕ)
iff $\exists i : 0 \leq i \leq n \,.\, s_i \models \phi$;

$\langle s_0, s_1, ..., s_n \rangle \models$ (`sometime-after` ϕ ψ)
iff $\forall i$. if $s_i \models \phi$ then $\exists j : i \leq j \leq n \,.\, s_j \models \psi$;

$\langle s_0, s_1, ..., s_n \rangle \models$ (`sometime-before` ϕ ψ)
iff $\forall i$. if $s_i \models \phi$ then $\exists j : 0 \leq j < i \,.\, s_j \models \psi$;

$\langle s_0, s_1, ..., s_n \rangle \models$ (`at-most-once` ϕ)
iff $\forall i \,.\, 0 \leq i \leq n$ if $s_i \models \phi$ then $\exists j : j \geq i, \forall k : k > j, s_k \models \neg\psi$;

where ϕ and ψ are first-order formulas and there is no nested modal operator.

■

There are some planners that natively support PDDL3. One of them is SGPlan [Hsu and Wah, 2008]. It solves a PDDL3 problem by partitioning a large planning problem into subproblems, each with its own goal, and resolves inconsistent solutions using the extended saddle-point condition. However, there is no sufficient information on how the problem can be effectively partitioned and solved. Its implementation is

closed source, while its binary distribution cannot solve problems other than IPC-5 domains[6].

Another planner that natively supports PDDL3 is MIPS-XXL [Edelkamp and Jabbar, 2008]. It compiles every state trajectory constraint into a Büchi automaton whose state can be updated by artificial actions. It uses Metric-FF [Hoffmann, 2003] as the base planner. During search, the base planner composes the actions to generate the solution plan, while also updating the state of the automatons by executing the artificial actions in order to maintain the state trajectory constraints. Unlike SGPlan, MIPS-XXL can solve problems other than IPC-5 domains.

Based on the same idea, [Baier and McIlraith, 2006] and [Gerevini et al., 2009] introduced techniques that compile a PDDL3 problem into a PDDL2 problem – they compile every state trajectory constraint into a finite automaton whose states are represented by artificial variables. Both techniques differ on the way to model the automaton's state transitions: the former uses axioms, while the latter modifies the original actions by adding extra conditional effects.

The main advantage of these compilation schemes is that any planner which supports PDDL2 can solve a PDDL3 problem without any modification. However, only some heuristics such as h^{FF} that supports a planning problem with axioms and conditional effects. Some good admissible heuristics such as h^{LM-Cut} [Helmert and Domshlak, 2009] and h^{LM} do not support axioms nor conditional effects.

This subsection introduces an alternative compilation scheme by adapting the technique described in §4.1. Similar with the above compilation schemes, every state trajectory constraint is represented as a finite automaton. However, the automaton transitions are represented using artificial actions, where no axiom and no conditional effect are required to model the automaton state transition. Thus, any heuristic that does not support axioms or conditional effects can be used to solve the problem using this compilation scheme.

The compilation scheme of the state trajectory constraint is summarised in the following paragraphs.

at-end

Expression (`at-end` ϕ) is just an alias of `goal` modal operator of PDDL2. Thus, the compilation is just straightforward ,that is by combining every `at-end` expression

[6]The automated planning community is holding a biennial planning competition called as International Planning Competition (IPC). IPC-5 domains is a set of planning problems used to benchmark the planners in the fifth International Planning Competition.

into a conjunction formula and then assign it as the goal of the compiled planning problem. For example, if $(\texttt{at-end}\ \phi_1),(\texttt{at-end}\ \phi_2),...,(\texttt{at-end}\ \phi_m) \in \mathcal{G}$ then it is converted to: $(\texttt{goal}\ (\texttt{and}\ \phi_1\ \phi_2\ ...\ \phi_m))$. Commonly, the planner further compiles the goal formula into lower level representation such as FDR or STRIPS by converting the formula to DNF. Each DNF clause becomes the precondition of an artificial action that makes an artificial predicate true, where the predicate is false at the initial state and true at the goal state. Thus, the planner can only reach the goal state if one of the clauses is satisfied.

always

Expression $(\texttt{always}\ \phi)$ has equivalent semantics to the global constraint i.e. formula ϕ must be always satisfied at every visited state. Thus, the technique described in §4.1 can be applied to this modal operator. An artificial predicate (`satisfy-always`) is introduced as the status flag of the formula – it is true if the formula is valid, otherwise false. The predicate is false at the initial state, since we assume that the initial state does not satisfy the formula. And it is true at the goal state. An artificial action `verify-always` is created to verify every visited state and defined as follows:

```
(:action satisfy-always
    (:precondition (and (not (satisfy-always)) φ )
    (:effect (satisfy-always)
)
```

If there are multiple instances of `always` formula, then they can be combined into a single formula so only one artificial predicate and one artificial action are required. For example, if $(\texttt{always}\ \phi_1),(\texttt{always}\ \phi_2),...,(\texttt{always}\ \phi_m) \in \mathcal{G}$ then $\phi = \phi_1 \wedge \phi_2 \wedge ... \wedge \phi_m$.

sometime

The interpretation of $(\texttt{sometime}\ \phi)$ means that ϕ must be satisfied in at least one state. An artificial action is introduced which must be selected by the planner at any point of the plan to ensure that the formula has been satisfied. w is a unique symbol to distinguish the expression from others, which can be a number. Assume that $w = 1$, then an artificial predicate `sometime-1` is introduced whose value is *false* at the initial state and *true* at the goal state. An artificial action `verify-1` is introduced and defined as follows:

```
(:action verify-sometime-1
    (:precondition (and (not (sometime-1)) φ )
```

```
    (:effect (sometime-1)
)
```

Unlike `always`, if there are multiple instances of `sometime` expression, then they would not be combined since every instance can be satisfied at different states. Thus, each instance must be handled independently with others.

at-most-once

Expression (`at-most-once` ϕ) is interpreted that ϕ may not be satisfied, or once it becomes true until a state is reached in which ϕ becomes and remains *false*. Based on this interpretation, two artificial predicates are required to represent the *true-period* and *post-true-period*. Assume w_1 and w_2 are unique symbols (they can be simply numbers) to as the flags of *true-period* and *post-true-period*. An artificial predicate (`at-most-once-`w_1) is introduced that represents the plan entering the *true-period* i.e. states $\langle s_i, s_{i+1}, ..., s_j \rangle$ where $\forall k \,.\, i \leq k \leq j \Rightarrow s_k \models \phi$. The predicate is false at the initial state and will become true whenever ϕ is true. Another artificial predicate (`at-most-once-`w_2) represents that the plan execution has entered the *post-true-period* i.e. states $\langle s_{j+1}, s_{j+2}, ..., s_n \rangle$ where $\forall l \,.\, (j+1) \leq k \leq n \Rightarrow s_k \models \neg\phi$. This predicate is false at the initial state. Since action `verify-always` is always executed after executing another action, then it can be used to update the predicates' value along the plan by adding two conditional effects which are:

1. (`when` ϕ (`at-most-once-`w_1)), which sets predicate (`at-most-once-`w_1) to be true whenever the plan enters the *true-period*;

2. (`when` (`and` (`at-most-once-`w_1) (`not` ϕ)) (`at-most-once-`w_2)) which sets predicate (`at-most-once-`w_2) to be true whenever the plan leaves the *true-period* and enters the *post-true-period*.

To avoid the plan entering the *true-period* for the second time, then the condition (`imply` (`at-most-once-`w_2) (`not` ϕ)) is added into the preconditions of action `verify-always`.

sometime-after

Interpretation of (`sometime-after` ϕ ψ) is that if ϕ is true at particular state then ψ must be true at the same or next state. An artificial predicate (`after-`w) is introduced to represent ψ may not be achieved. Note that w is a unique symbol which can be a number. It is true at the initial state and goal. Based on the interpretation,

a conditional effect (when (and ϕ (not ψ)) (not (after-w))) is added to action verify-always so that whenever ϕ is true and ψ is false after execution, then the artificial predicate is set false in order to enforce the plan to achieve ψ at the next state, since the goal requires predicate (after-w) to be true. Another conditional effect (when (and ψ (not (after-w))) (after-w)) is added that sets the predicate true whenever ψ has been achieved.

sometime-before

Expression (sometime-before ϕ ψ) means that whenever ϕ is true then ψ must be true at the previous state. An artificial predicate (before-w) is introduced to represent a condition that ψ is true at particular state. Note that w is a unique symbol which can be a number. The predicate is false at the initial state. To update the predicate, conditional effect (when ψ (before-w)) is added to action verify-always that sets the predicate true whenever ψ is true. Since ϕ may not be achieved unless ψ has been achieved, then condition (imply ϕ (before-w)) is added to the precondition of action verify-always.

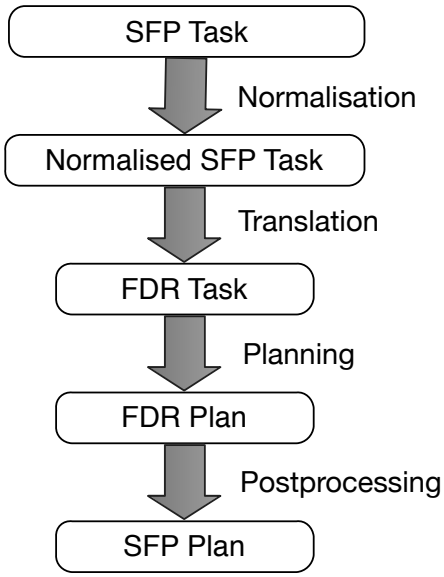


Figure 4.3: Overview of the steps for solving SFP Task

4.2 Configuration Task as Classical Planning Problem

A configuration task is the problem of determining the workflow that can bring the system from one to another configuration state. §3.3 has shown an example of a configuration state that can be described in SFP. Thus, we can formally define the configuration task as the following.

Definition 4.5 (SFP Task).
An SFP configuration task is a 3 tuple $\Sigma = \langle \sigma_0, \sigma_g, \Gamma \rangle$, where $\sigma_0 \in \mathcal{S}$ is the *current* configuration state of the system, $\sigma_g \in \mathcal{S}$ is the *desired* configuration state of the system, and Γ is the type environment that holds the type of every element in σ_0 and σ_g. ■

An SFP task can be solved by compiling it into an FDR task, and then can be given as the input of a classical planner for finding a solution plan. Figure 4.3 illustrates the overview of the steps to solve an SFP task. The details of all steps are described in the following subsections.

4.2.1 Normalisation

The aim of normalisation of the SFP task $\Sigma = \langle \sigma_0, \sigma_g, \Gamma \rangle$ is to flatten the tree structure of σ_0 and σ_g. This is because the variables in SFP are organised in a tree-structure (see

definition 3.5 in §3.2.2.1), while the variables in FDR (as the target representation) are organised in a flat-structure.

Definition 4.6 (Flat-Store).
A *flat-store* is a set of reference-value pairs $\Omega = \{\ \omega_i \mid \omega_i = (\mathcal{R} \times \mathcal{V})\ \}$. It is the flat version of store $\sigma \in \mathcal{S}$. If $\omega = \langle r, \upsilon \rangle$, then we call r as the SFP name and υ as the SFP value. Every reference r_i can be classified as follows:

- r_i is a *prevail* reference of Ω
  iff $\langle r_i, \upsilon_i \rangle \in \Omega$
- r_i is a *nested* reference of Ω
  iff $\langle r_i, \upsilon_i \rangle \notin \Omega$ and $\exists j \,.\, r_j \subset_{\mathcal{R}} r_i \wedge \langle r_j, \upsilon_j \rangle \in \Omega$
- r_i is an *invalid* reference of Ω
  iff r_i is neither *prevail* nor *nested*

■

Based on the above definition, we then can define the *normalised* SFP task which is the product of the normalisation process.

Definition 4.7 (Normalised SFP Task).
A *normalised* SFP task is a 3 tuple $\Lambda = \langle \Omega_0, \Omega_g, \Gamma \rangle$, where Ω_0 is the *normalised current* configuration state, Ω_g is the *normalised desired* configuration state, and Γ holds the type of every element in Ω_0 and Ω_g. ■

Definition 4.8 (Function *normalise*).
Function ***normalise*** converts store $\sigma \in \mathcal{S}$ to a normalised store Ω by visiting every element of σ and its children stores. If $r \in \mathcal{R}$ and $\upsilon \in \mathcal{V}$ are the element's absolute path and value of σ, then a pair $\langle r, \upsilon \rangle$ is added to Ω.

$$
\begin{aligned}
&\mathit{normalise} : \mathcal{S} \rightarrow \mathcal{P}(\Omega) \\
&\quad \mathit{normalise}(\sigma) \quad := \quad \mathit{normalisestore}(\sigma, \varnothing_{\mathbb{I}}) \\
&\mathit{normalisestore} : \mathcal{S} \times \mathcal{R} \rightarrow \mathcal{P}(\Omega) \\
&\qquad \mathit{normalisestore}(\varnothing_{\mathcal{S}}, ns) \quad := \quad \{\} \\
&\quad \mathit{normalisestore}(\langle id, v \rangle :: s, ns) \quad := \\
&\qquad \text{if } v \in \mathcal{S} \text{ then } \{\langle r, \varnothing_{\mathcal{S}} \rangle\} \cup \mathit{normalisestore}(v, r) \cup \mathit{normalisestore}(s, ns) \\
&\qquad \text{else } \{\langle r, v \rangle\} \cup \mathit{normalisestore}(s, ns) \\
&\text{where } r = ns \oplus id
\end{aligned}
$$

■

Example.
Let $\sigma = \langle \mathtt{a}, \langle \mathtt{c}, 3\rangle :: \varnothing_{\mathcal{S}}\rangle :: \langle \mathtt{b}, \varnothing_{\mathcal{S}}\rangle :: \varnothing_{\mathcal{S}}$, then
$normalise(\sigma) = \{\langle \mathtt{a} :: \varnothing_{\mathbb{I}}, \varnothing_{\mathcal{S}}\rangle, \langle \mathtt{a} :: \mathtt{c} :: \varnothing_{\mathbb{I}}, 3\rangle, \langle \mathtt{b} :: \varnothing_{\mathbb{I}}, \varnothing_{\mathcal{S}}\rangle\}$

To get Λ, σ_0 and σ_g must be converted to Ω_0 and Ω_g respectively as follows:

$$\begin{aligned} \Omega_0 &= normalise(\sigma_0) \\ \Omega_g &= normalise(\sigma_g) \end{aligned}$$

4.2.2 Translation

The translation process aims to convert a normalised SFP task $\Lambda = \{\Omega_0, \Omega_g, \Gamma\}$ to an FDR task $\Pi = \langle V, A, s_0, s_g\rangle$. The translation is done in six steps. The first five steps generate the variables V, the actions A, the initial state s_0, the goal s_g, and the global constraint formulae S_β respectively. These are the components of the planning problem with global constraint Θ (see definition 4.1). The final step compiles Θ to classical planning Π using the technique described in §4.1. The details of all steps are described in the following paragraphs.

Step 1 (Variables) Generate A (FDR variables) by selecting every element in current state whose type is neither an action nor a global constraint. The element's type can be found in Γ. The variable's value domain is determined by collecting all values in the current (Ω_0) and desired (Ω_g) state whose type is a subtype of the variable's type. Every value in the precondition and effect of SFP action whose type is subtype of the variable's type, should be added as well to the variable domain.

Step 2 (Global Constraint) Determine the global constraint formulae by searching for an element $\langle r_\beta, \upsilon_\beta\rangle \in \Omega_g$ where $r_\beta = \mathtt{global} :: \varnothing_{\mathbb{I}}$ and $\upsilon_\beta \in \mathbb{C}$ is the global constraints formula. The formula must be processed as follows:

1. Every *nested*[7] reference must be rewritten into an existential quantification of conjunction over *prevail* references such that there is no nested reference in the final formula. Without loss of generality, this process can be illustrated using the example specification in figure 3.11. In line 16, the constraint has a conjunction clause with a nested reference i.e. `pc1.refer.running = true`. Since there are two objects that has type `Service` i.e. `s1` and `s2`, then the clause should be rewritten as:

[7] See definition 4.6.

$$(\texttt{pc1.refer} \neq \texttt{null}) \wedge$$
$$((\texttt{pc1.refer} = \texttt{s1.web}) \Rightarrow (\texttt{s1.web.running} = \texttt{true})) \wedge$$
$$((\texttt{pc1.refer} = \texttt{s2.web}) \Rightarrow (\texttt{s2.web.running} = \texttt{true}))$$

2. Reduce the variable domain's members based on *MemberOfList* constraints. For example, since the variable `pc1.refer` has type `*Service` (reference of Service), then its domain is $\{\texttt{s1}, \texttt{s2}, \texttt{null}\}$. However, based on the above global constraints, then value `null` shoud be removed from the domain. Thus, the final domain of `pc1.refer` is $\{\texttt{s1.web}, \texttt{s2.web}\}$. And since `null` has been removed from the variable domain, then clause $(\texttt{pc1.refer} \neq \texttt{null})$ can be safely removed from the global constraints formula.

The result of the above process is a logic formula over SFP atoms, where each SFP atom is an assignment in the form of $r = \upsilon$. Since r refers to the name of variable $v \in V$, then the formulae can be converted to a logic formulae over FDR atoms since r can be replaced by variable v. This formula is then set to S_β.

Step 3 (Actions) Generate the set of actions $A = \{a_1, ..., a_m\}$ by collecting every element $\langle r_a, \upsilon_a \rangle \in \Omega_0$ where $(r : \texttt{act}) \in \Gamma$, and $\upsilon \in \mathbb{A}$ is an SFP action. Every SFP action must be grounded by substituting every parameter with the value whose type is subtype of the parameter's type. After substitution, every *nested* reference in the precondition must be rewritten so that all references in the precondition are *prevail* references. This can be done using the scheme described in step 2. And then the precondition formula is converted into a Disjunction Normal Form (DNF) where for each conjunction clause, the original action is copied and the new action's precondition is replaced by the conjunction clause. Afterwards, the original action is then discarded[8]. This process yields a set of actions, each of which is an SFP *grounded action*.

Definition 4.9 (SFP Grounded Action).
An SFP grounded action is a 5-tuple $\hat{a} = \langle \hat{name}, \hat{cost}, \hat{params}, \hat{pre}, \hat{eff} \rangle$, where:

- $\hat{name} \in \mathcal{R}$, is the action name;
- $\hat{cost} \in \mathbb{R}^{0+}$, is a non-negative action cost defined in SFP by the user;
- $\hat{params} = \{par_i \mid par_i = \langle id, x \rangle, id \in \mathbb{I}, x \in \mathbb{V}\}$, is the set of parameters;
- $\hat{pre}$ and $\hat{eff}$, are conjunctions over SFP atoms, called as precondition and effect respectively.

[8]The precondition is true iff any conjunction clause is true.

■

Note that the parameters and their value are still kept since they will be passed to the software component during execution.

Finally, every SFP grounded action is converted to the FDR action $a = \langle name, cost, pre, eff \rangle$ such that:

- $\hat{name}$ and $\hat{params}$ are encoded as string literal[9], and then it is assigned to $name$,
- $cost = \hat{cost}$,
- for every SFP atom $r = \upsilon$ in $\hat{pre}$, then $pre(v) = \upsilon$ iff r is the name of variable v, and
- for every SFP atom $r = \upsilon$ in $\hat{eff}$, then $eff(v) = \upsilon$ iff r is the name of variable v.

Step 4 (Initial State) Generate the initial state s_0 as variable assignment over $V = \{v_1, ..., v_n\}$ such that $s_0(v_i) = \upsilon_i$ iff $\langle r_i, \upsilon_i \rangle \in \Omega_0$ and r_i is the name of variable v_i.

Step 5 (Goal) Generate the goal s_g as variable assignment over $V = \{v_1, ..., v_n\}$ such that $s_g(v_i) = \upsilon$ iff $\langle r_i, \upsilon_i \rangle \in \Omega_g$ and r_i is the name of variable v_i;

Step 6 (Compile Θ to Π) Steps 1-5 generates components of $\Theta = \langle V, A, s_0, s_g, S_\beta \rangle$, which is the planning problem with global constraint (see definition 4.1). Thus, the FDR task $\Pi = \langle V, A, s_0, s_g \rangle$ can be generated by calling function COMPILE-GLOBAL (see algorithm 4.1) where:

$$\Pi = \text{COMPILE-GLOBAL}(V, A, s_0, s_g, S_\beta)$$

4.2.3 Planning

Planning involves finding the solution plan by giving the FDR task Π as the input of the classical planner. Any planner that supports FDR can be used to solve the task. In the planning step, we employ a *multi-heuristics* and *two-stages* strategy that aims to increase the coverage of the planner as well as to generate the suboptimal plan under given deadline[10].

The multi-heuristics strategy is based on the fact of International Planning Competition (IPC) results that there is no heuristic which is dominant on every problem

[9] The string literal will be decoded later to convert an FDR plan to an SFP plan.

[10] In practice, the global optimal plan is not the priority in particular when a reconfiguration must be performed as soon as possible to address a failure.

domain. In the sense that the heuristic is sensitive to the characteristics of the problem. Thus, in order to increase the probability of finding the plan, multiple heuristics can be used in parallel by running several planner instances, each of which is using particular heuristic. This can be implemented using a planner that supports multiple heuristics, or several planners that have different heuristics.

There are two types of heuristic. First is the "admissible" heuristic which can be used with A* search algorithm to find a global optimal plan. Second is the "inadmissible" heuristic which can be used with a greedy search algorithm to find a satisficing plan – the plan is not guaranteed to be global optimal[11]. However, the admissible heuristic with A* search is naturally much slower than inadmissible with greedy search since finding an optimal plan is much more difficult than finding a satisficing plan. In the two-stages strategy, we combine them by employing an inadmissible heuristic at the first stage in order to find the plan as soon as possible although it is not global optimal. Afterwards, an admissible heuristic is employed at the second stage where the planning problem has been shrunk by eliminating non-artificial actions which are not selected at the first stage. The aims of the second stage is to optimise the plan found at the first stage.

In implementation, combining these two strategies is straightforward where multiple planner instances are running in parallel, each of which is using particular inadmissible heuristic with greedy search at the first stage. Whenever an instance has found a plan, it will shrink the planning problem and then restart the search using admissible heuristic with A* search. All searches will stop whenever the deadline has been reached.

4.2.4 Post-processing

The post-processing has two main objectives. First is converting the total-order plan generated by the planner to a partial-order plan. This will enable parallel execution which could decrease the execution time. Second is converting every FDR action back to SFP action. This reverse-conversion is required since the deployment system is using SFP representation to perform particular task (see §5 for more details). For example, the controller will use the reference of the SFP action to determine which agent should be requested in order to execute a particular action.

Assume π is an FDR plan generated by the planner. Then the post-processing

[11] In classical planning, the optimality of a plan is equal to the total cost of all actions. Hence, the plan is global optimal iff there is no other plan that has less cost.

is performed by calling function GENERATE-PARTIAL-ORDER (see definition 4.2) in order to remove the artificial actions and generate the partial-order plan π_{po}, i.e.:

$$\pi_{po} = \{A_{po}, \prec\} = \text{GENERATE-PARTIALORDER}(\pi)$$

where A_{po} is a set of actions of the particular order plan, $\prec$ is a set of partial ordering constraints between the actions.

Definition 4.10 (SFP Plan).
An SFP plan is a 2-tuple $\pi_{SFP} = \langle \hat{A}, \prec \rangle$, where:
$\hat{A} = \{\hat{a}_i \mid \hat{a}_i \text{ is an SFP grounded action}\}$. ■

Afterwards, π_{po} is converted to SFP plan π_{SFP} by converting every FDR action $a \in A_{po}$ to SFP grounded action $\hat{a}$. This is the opposite process of step 3 of §4.2.2.

4.2.5 Example

This subsection gives an example of the above processes to solve an SFP configuration task described in §3.3. Note that for brevity, a reference is written in an informal form, for example: a reference `a:b:`$\varnothing_{\mathbb{I}}$ is written as `a.b`.

4.2.5.1 Normalisation

Using function *normalise*, the specifications of the current and the desired states (see figure 3.10 and 3.11) were converted to flat stores as described in the following table.

r_i	$\upsilon_i \in \Omega_0$	$\upsilon_i \in \Omega_g$	$\tau_i \in \Gamma$
s1	object	object	Machine
s1.dns	"ns.foo"	"ns.foo"	str
s1.web	object	object	Service
s1.web.running	true	false	bool
s1.web.port	80	80	num
s2	object	object	Machine
s2.dns	"ns.foo"	"ns.foo"	str
s2.web	object	object	Service
s2.web.running	false	true	bool
s2.web.port	80	80	num
pc1	object	object	Client
pc1.refer	s1.web	s2.web	*Service
pc2	object	object	Client
pc2.refer	s1.web	s2.web	*Service

Note that the first column contains the variable references, the second column contains the value of Ω_0 (current value), the third column contains the value of Ω_g (desired value), and the fourth column contains the variables' type.

4.2.5.2 Translation

The first step of translation extracts variables from Ω_0 and then assigns a domain to every variable. This yielded the following variables where the left hand-side is the variable name and the right is the variable domain.

r_i	D_{r_i}
s1	{object}
s1.dns	{"ns.foo"}
s1.web	{object}
s1.web.running	{true, false}
s1.web.port	{80}
s2	{object}
s2.dns	{"ns.foo"}
s2.web	{object}
s2.web.running	{true, false}
s2.web.port	{80}
pc1	{object}
pc1.refer	{null, s1.web, s2.web}
pc2	{object}
pc2.refer	{null, s1.web, s2.web}

The second step of translation rewrites the nested references in the global constraints formula. After rewriting, the formula will be:

$$
\begin{aligned}
&(\texttt{pc1.refer} \neq \texttt{null}) \wedge \\
((\texttt{pc1.refer} = \texttt{s1.web}) &\Rightarrow (\texttt{s1.web.running} = \texttt{true})) \wedge \\
((\texttt{pc1.refer} = \texttt{s2.web}) &\Rightarrow (\texttt{s2.web.running} = \texttt{true})) \wedge \\
&(\texttt{pc2.refer} \neq \texttt{null}) \wedge \\
((\texttt{pc2.refer} = \texttt{s1.web}) &\Rightarrow (\texttt{s1.web.running} = \texttt{true})) \wedge \\
((\texttt{pc2.refer} = \texttt{s2.web}) &\Rightarrow (\texttt{s2.web.running} = \texttt{true}))
\end{aligned}
$$

Based on the above global constraints, the domains of pc1.refer and pc2.refer can be reduced by removing null. Afterwards, clause (pc1.refer $\neq$ null) and (pc2.refer $\neq$ null) can be safely removed from the global constraints formula. Thus, the final variable domains are:

r_i	D_{r_i}
s1	{object}
s1.dns	{"ns.foo"}
s1.web	{object}
s1.web.running	{true,false}
s1.web.port	{80}
s2	{object}
s2.dns	{"ns.foo"}
s2.web	{object}
s2.web.running	{true,false}
s2.web.port	{80}
pc1	{object}
pc1.refer	{s1.web,s2.web}
pc2	{object}
pc2.refer	{s1.web,s2.web}

And the final global constraints formula is:

$$((\texttt{pc1.refer} = \texttt{s1.web}) \Rightarrow (\texttt{s1.web.running} = \texttt{true})) \wedge$$
$$((\texttt{pc1.refer} = \texttt{s2.web}) \Rightarrow (\texttt{s2.web.running} = \texttt{true})) \wedge$$
$$((\texttt{pc2.refer} = \texttt{s1.web}) \Rightarrow (\texttt{s1.web.running} = \texttt{true})) \wedge$$
$$((\texttt{pc2.refer} = \texttt{s2.web}) \Rightarrow (\texttt{s2.web.running} = \texttt{true}))$$

The third step of translation is grounding the SFP actions and then converting them to FDR actions. There are 6 SFP actions i.e. `s1.web.start`, `s1.web.stop`, `s2.web.start`, `s2.web.stop`, `pc1.redirect`, and `pc2.redirect`. By substituting the parameters, the grounding and the convertion processes will yield 8 FDR actions, which are:

- name: `s1.web.start()`
  pre: `s1.web.running=false`
  eff: `s1.web.running=true`
- name: `s1.web.stop()`
  pre: `s1.web.running=true`
  eff: `s1.web.running=false`
- name: `s2.web.start()`
  pre: `s2.web.running=false`
  eff: `s2.web.running=true`

- name: `s2.web.stop()`
  pre: `s2.web.running=true`
  eff: `s2.web.running=false`
- name: `pc1.redirect(s=s1.web)`
  pre: `s1.web.running=true`
  eff: `pc1.refer=s1.web`
- name: `pc1.redirect(s=s2.web)`
  pre: `s2.web.running=true`
  eff: `pc1.refer=s2.web`
- name: `pc2.redirect(s=s1.web)`
  pre: `s1.web.running=true`
  eff: `pc2.refer=s1.web`
- name: `pc2.redirect(s=s2.web)`
  pre: `s2.web.running=true`
  eff: `pc2.refer=s2.web`

The fourth and the fifth steps of translation are direct conversions from Ω_0 and Ω_g of SFP to s_0 and s_g of FDR respectively.

In the sixth step of translation, the FDR task will be compiled into a classical planning problem using the compilation technique described in §4.1. Since the final global constraints is a conjunction of simple implications, then we can use the simple-implication compilation rules of definition 4.3 to compile the constraints. This will yield the following FDR actions:

- name: `s1.web.start()`
  pre: `s1.web.running=false` $\wedge$ `pc1.refer= s2.web` $\wedge$
  `pc2.refer=s2.web` $\wedge$ `s2.web.running=true`
  eff: `s1.web.running=true`
- name: `s1.web.stop()`
  pre: `s1.web.running=true` $\wedge$ `pc1.refer= s2.web` $\wedge$
  `pc2.refer=s2.web` $\wedge$ `s2.web.running=true`
  eff: `s1.web.running=false`
- name: `s2.web.start()`
  pre: `s2.web.running=false` $\wedge$ `pc1.refer= s1.web` $\wedge$
  `pc2.refer=s1.web` $\wedge$ `s1.web.running=true`
  eff: `s2.web.running=true`

- name: `s2.web.stop()`
  pre: `s2.web.running=true` $\wedge$ `pc1.refer= s1.web` $\wedge$
  `pc2.refer=s1.web` $\wedge$ `s1.web.running=true`
  eff: `s2.web.running=false`
- name: `pc1.redirect(s=s1.web)`
  pre: `s1.web.running=true` $\wedge$ `s1.web.running=true` $\wedge$
  `s2.web.running=true`
  eff: `pc1.refer=s1.web`
- name: `pc1.redirect(s=s2.web)`
  pre: `s2.web.running=true` $\wedge$ `s1.web.running=true` $\wedge$
  `s2.web.running=true`
  eff: `pc1.refer=s2.web`
- name: `pc2.redirect(s=s1.web)`
  pre: `s1.web.running=true` $\wedge$ `s1.web.running=true` $\wedge$
  `s2.web.running=true`
  eff: `pc2.refer=s1.web`
- name: `pc2.redirect(s=s2.web)`
  pre: `s2.web.running=true` $\wedge$ `s1.web.running=true` $\wedge$
  `s2.web.running=true`
  eff: `pc2.refer=s2.web`

Note that the artificial variable (v_β) and action (a_β) are not added to the final problem since all conjunction clauses are simple implications.

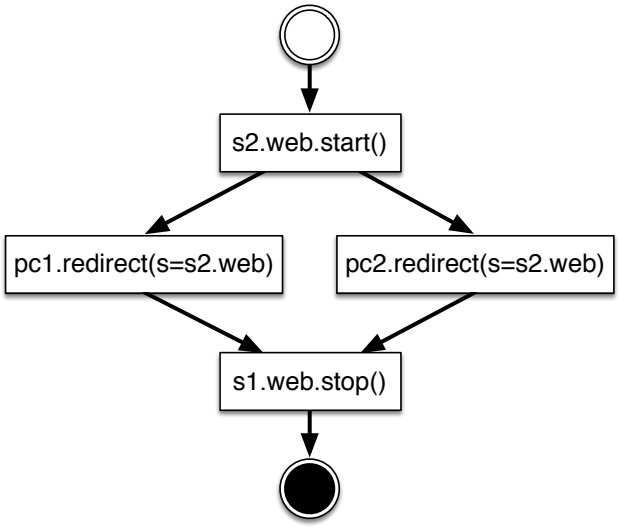
4.2.5.3 Planning

By giving the FDR task to a classical planner, then the following sequential plan will be generated:

`s2.web.start()` $\rightarrow$ `pc1.web.redirect(s=s2.web)` $\rightarrow$
`pc2.web.redirect(s=s2.web)` $\rightarrow$ `s1.web.stop()`

4.2.5.4 Postprocessing

Finally, the postprocessing will yield the following partial-order plan:



4.2.6 Loose Specification

A "loose" specification is very useful to overcome failures since the tool can automatically select an alternative state from a set of possible final states, and then generate and execute a workflow to transition from the current to the selected final state.

To generate a workflow that can achieve one of these states, the constraints formula of the final state should be converted into a DNF formula. If the DNF after conversion is in the form of $\psi_1 \vee ... \vee \psi_n$ where ψ_i is a conjunction of atoms, then an artificial action a^i_γ is introduced and ψ_i is set as its preconditions. An artificial variable v_γ is introduced whose value is false at the initial state, and true at the goal state. Finally, all artificial actions will have an effect that sets variable v_γ from false to true. This will force the planner to select an artificial action in order to achieve the goal state, where the preconditions of the selected artificial action is the selected final state.

However, generating all possible final states might not efficient, in particular when the number of possible states are large. Thus, we might be only need to generate n final states from m possible solutions, where $n \ll m$ (n is much less than m). Unfortunately, there is no guarantee that there is a workflow which can bring the system from the current to any of these n states. We leave the answer of this issue as part of the future works.

4.3 Summary

This chapter has presented a domain-independent technique to compile a planning problem with extended goals into a classical planning problem. The main advantage of this compilation is that we can use any classical planner to solve the problem without any modification. And since the automated planning community is still actively developing new techniques for solving classical planning problems, then we use these new techniques to improve the performance of the planners.

The second part of this chapter presented a technique that translates a configuration task defined in SFP language into a classical planning problem in FDR encoding. This enables us to use the off-the-shelf classical planner, such as FastDownward [Helmert, 2006], for solving configuration tasks. To improve the performance of the planner, we can use multi-heuristics and two-stage strategy to increase the planning coverage as well as improving the plan optimality under given deadline. The total order plan which is generated by the classical planner can then be processed further to generate a partial order plan.

Chapter 5

Deploying Configuration Changes

Deployment is the process that implements configuration changes to bring the system from the current to the desired state while preserving the global constraint. This can be done in several ways. First is the orchestration technique where one agent acts as the planner and the execution orchestrator of others. Although it is fairly simple to be implemented, but this fully centralised architecture creates a potential bottleneck in large systems, and can also be unreliable if the communications with the controller is disrupted, which is particular relevant since reconfiguration frequently occurs as an autonomic response to system failures.

On the other hand, we can use a fully distributed system where each agent can make its own decision by independently planning and executing the workflow to achieve its goal. It is scalable and there is no single point of failure. However, it is not a good solution to the above problem either. Avoiding deadlock or livelock situation may require agents to have considerable global knowledge. Achieving such knowledge is likely to result in even more costly inter-agent communication. Predicting the behaviour of such systems is also more difficult and hence they are less acceptable to system administrators in real situations.

The main contribution of this chapter is providing a solution which aims to avoid the shortcomings of two extremes: fully centralised and fully distributed approaches. This solution is mixed: the workflow is generated centrally by an agent, which is then used to automatically construct a set of purely reactive agents which *choreograph* the execution of the workflow without the need for a central controller. This combination of centralised planning with distributed execution provides robust, autonomous execution while retaining advantages of a predictable, deadlock-free workflow. In addition, using a regression algorithm of workflow execution enables the agents to form

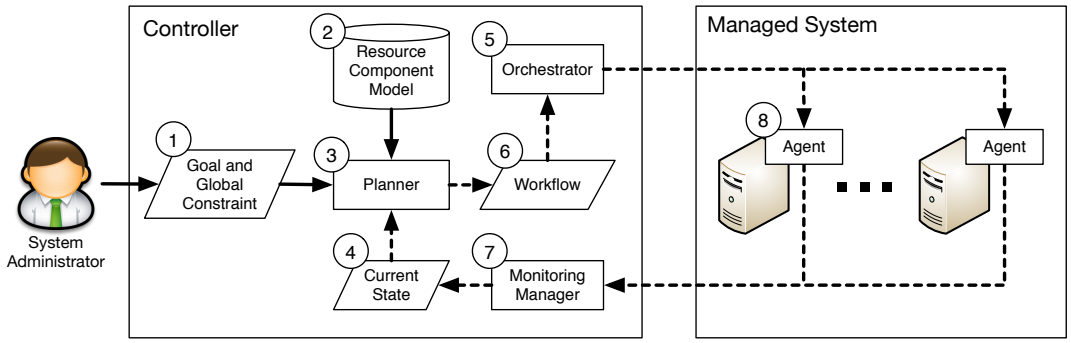


Figure 5.1: The orchestration architecture.

a self-healing system by continuously attempting to maintain the goal state.

The chapter starts by giving details of the orchestration architecture with a closed-loop control. It is followed by a section that presents the "choreography" technique that constructs and executes the model of reactive agents. Finally, we summarise this chapter in the last section.

5.1 Orchestration

In system configuration, orchestration is a deployment technique where an agent becomes the central controller of the system. It is acting as the planner which generates the workflow, and the orchestrator which schedules the actions execution on other agents in order to enforce the ordering constraint of the workflow. Other agents are passive in the sense that they only perform a task requested by the controller. Figure 5.1 illustrates a typical orchestration architecture which mainly consists of four components: the planner, the orchestrator, the monitoring manager, and the agent. The relationships between the components can be described as follows[1]:

- The action database (2) holds a set of *resource component* models which would normally be defined by an expert, system engineer, software engineer or other specialist. Each model has a set of attributes that hold the state of a particular resource and a set of actions that can be executed to change the resource's state.

- Each managed node has an agent (8) that manages and monitors all resource components. The monitoring manager (7) in the controller periodically pulls

[1]Each number represents the component in figure 5.1.

and aggregates the current state of all nodes by sending the request to every agent in order to generate the current state of the system.

- System administrator specifies the goal state and the global constraint (1) that should be achieved by the system. This specification together with the resource component models and the current state are then used by the planner (3) to generate the workflow.

- To deploy the specification, the orchestrator (6) orchestrates the execution of the workflow by scheduling the action that should be executed by target agent at particular time.

It is possible for a failure to occur during the execution of a workflow. This architecture could implement a pragmatic approach to handling such failures. Each agent is responsible for ensuring that each action is executed successfully. In addition, it also has to detect and report any failure to the orchestrator. If such failure is reported, then the execution of the workflow is immediately discontinued. The orchestrator will then send a request to the planner to generate an alternative workflow for later execution. A notification may also be shown in the user interface so that the administrator is informed of the failure, and can submit an alternative goal and global constraint, if required.

This architecture could be used as a fully-automated "unattended" configuration tool which automatically corrects any drift in the specification without any human intervention. In figure 5.1, the dashed arrow lines show a loop process which could be set to be activated periodically by the administrator in order for the system to verify and correct any drift of the current state from the goal such as outdated software packages, accidentally stopping a service, etc.

Figure 5.2 shows the architecture of an agent that controls a particular machine. It has a daemon (3) that sends the state (1) to and receives the action (2) from the controller. Based on particular specification, it automatically constructs a set of resource components, each of which is the software component that controls a particular resource such as a file or a service.

An important feature of this architecture is that there is a clear separation between the "configuration" (4) and the "implementation" (5) of the agent and the resource component. This loose coupling makes the configuration *platform independent*. This opens up the possibility to implement the agent and the resource component in different platforms while they use the same configuration representation, for example SFP.

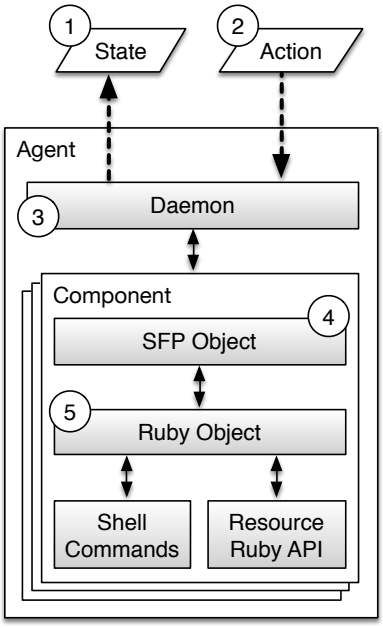


Figure 5.2: The architecture of an agent where the deployment part of the resource component is implemented in Ruby.

Orchestration is fairly simple to implement and operate since it is similar to the typical organisation structure where every decision is made centrally. We can easily predict the behaviour of the whole system because every configuration change is based on a *deterministic* workflow, which can be reviewed and approved before execution. In practice, a predictable system is more acceptable to system administrator than another. In addition, the communication cost is not significant because every agent only requires to communicate with the controller. It does not need to contact other agents to have the global knowledge since every global conflict will be resolved by the controller.

However, this simplicity comes at a price: the controller agent becomes the single-point of failure. Any unreliable communication or hardware can disrupt the operation of the whole system. In addition, the controller needs to plan the workflow for every changes, even the small ones, where the global knowledge must be taken into account on every planning. This is relatively inefficient for "repetitive" reconfiguration where some parts of the current plan could be reused to achieve the goal. These problem will be solved with the choreography technique described in the next section.

5.2 Choreography

Choreography is the process to define a "global scenario" that should be executed by every agent to implement particular specification. The global scenario is the global plan generated by the planner. If the plan involves only a single agent, then the execution can be performed in a straightforward way. However, if the scenario involves more than one agent, then it must be split up into a set of "local scenarios", each of which is for an agent. We refer to one local scenario as a *local Behavioural Signature* (BSig) model which defines the local goal that should be achieved by the agent, and specifies *which* local changes (actions) can be made *under what circumstances*. We refer the set of all local BSig models as the *global BSig* model.

In choreography, the configuration deployment is divided into two parts. First is *choreographing* where from a set of agents, one of them[2] acts as a *choreographer* that aggregates the current states of other agents, does the planning to generate the global plan, and then uses the plan to construct the set of local BSig models, each of which is sent to a particular agent. Second is *execution* where every agent executes the local BSig model using a *regression* technique by finding and immediately executing the necessary actions that can repair any flaw in the local goal or the action's precondition. Every agent will perform peer-to-peer communications with particular agents, as specified in its BSig model, to request the necessary *remote* preconditions before executing an action. Whenever an agent cannot repair a particular flaw due to some reasons, such as there is no action that can repair the flaw or the supporting action's preconditions cannot be satisfied, then a *re-choreographing* must be done to generate a new global plan.

The BSig model can be *transient* or *persistent*. A transient model will be removed whenever the agents have attained their local goal. On the other hand, the persistent model will still be kept although the goal has been achieved. The combination of the persistent models and the regression algorithm of workflow execution enable the agents to form a *self-healing* system by periodically executing the models. This is possible because the regression execution makes the plan become valid when the system is either at initial or at any intermediate state of the plan. Hence, the agents can perform an "immediate response" to correct particular drifts from the desired state without the need of re-choreographing (replanning).

[2]The choreographer can be dynamically selected from the available agents. This dynamic selection is not covered in this thesis.

5.2.1 Assumptions

This choreography technique is based on several assumptions. First, the managed system has a set of resources, each of which has a set of variables that represent its state. Every agent has a set of actions can change the state of particular resource. Based on the actions, the variables can be classified into two types: *local* and *global* variables. A variable is *local* if only a single agent that can change its value. On the other hand, a variable is *global* if its value can be modified by more than one agent. Based on experiences, most of the variables are classified into the first type while only few of them are classified to the latter. The classification can be formalised as the following paragraph.

Let $\hat{A} = \{\hat{A}_i\}_{i=1}^{n}$ be the set of all actions, and $\hat{V} = \{\hat{V}_i\}_{i=1}^{n} \cup \hat{V}_g$ is the set of all variables of the system, where:

- n is the total number of agents;
- $\hat{A}_i$ and $\hat{V}_i$ are the set of actions and local variables of agent i respectively. $\forall i, j$. if $i \neq j$ then $\hat{V}_i \cap \hat{V}_j = \{\}$ and $\hat{A}_i \cap \hat{A}_j = \{\}$.
- $\hat{V}_g$ is the set of global variables.

Assume $\hat{a}_{i,k} \in \hat{A}_i$ and $\hat{a}_{j,l} \in \hat{A}_j$ are the actions of agent i and j respectively. Then the rules of variable classification are:

- Variable $\hat{v}$ is global iff $\exists i, j$. if $i \neq j$ then $\hat{v} \in \mathit{Effects}(\hat{a}_{i,k})$ and $\hat{v} \in \mathit{Effects}(\hat{a}_{j,l})$;
- Variable $\hat{v}$ is local of agent i iff $\hat{v} \in \mathit{Effects}(\hat{a}_{i,k})$ and $\nexists j$. if $i \neq j$ then $\hat{v} \in \mathit{Effects}(\hat{a}_{j,l})$.

The second assumption is that the SFP language is used to model the configurations of the system. For simplicity, the agents are organised in a flat structure where each of them has a unique identifier $id \in \mathbb{I}$. Since the name of every SFP action is a reference i.e. a list of identifiers, then its first identifier is used to identify the agent that owns the action. For example, an action has name `s1.web.start`, then agent `s1` is owner of the action.

Consider the example of multi-services system illustrated in figure 3.1 whose current state is given in listing 3.10. This specification of current state shows that the system has four agents i.e. `s1`, `s2`, `pc1`, and `pc2`, where the classification of variables is:

global variables	:	{}
s1's local variables	:	{s1.dns, s1.web.running}
s2's local variables	:	{s2.dns, s2.web.running}
pc1's local variables	:	{pc1.refer}
pc2's local variables	:	{pc2.refer}

For comparison, consider a simple cloud system whose current state is described in the following specifications:

```
1  // file: schemata.sfp (the resource models)
2  schema Cloud {
3    running: Boolean
4    def create_vm(vm: VM) {
5      condition { this.running = true; vm.in_cloud = null; }
6      effect    { vm.in_cloud  = this;                     }
7    }
8    def delete_vm(vm: VM) {
9      condition { this.running = true; vm.in_cloud = this; }
10     effect    { vm.in_cloud  = null;                     }
11   }
12 }
13 schema VM {
14   in_cloud : Cloud = null
15 }
```

```
1 include "schemata.sfp"
2 // current state
3 main {
4   cloud1 isa Cloud { running  = true;    }
5   cloud2 isa Cloud { running  = true;    }
6   vm1 isa VM       { in_cloud = cloud1; }
7 }
```

This system has three agents which are `cloud1`, `cloud2`, and `vm1`. The first two are managing the cloud infrastructures, while the latter is managing a virtual machine which can be on one of the clouds. There is a single global variable that represents the existence of `vm1` i.e. `vm1.in_cloud` which can be modified either by `cloud1` or `cloud2`. The other two variables i.e. `cloud1.running` and `cloud2.running` are local variables of `cloud1` and `cloud2` respectively. Hence, the classification of variables of this system is:

global variables	:	{vm1.in_cloud}
cloud1's local variables	:	{cloud1.running}
cloud2's local variables	:	{cloud2.running}

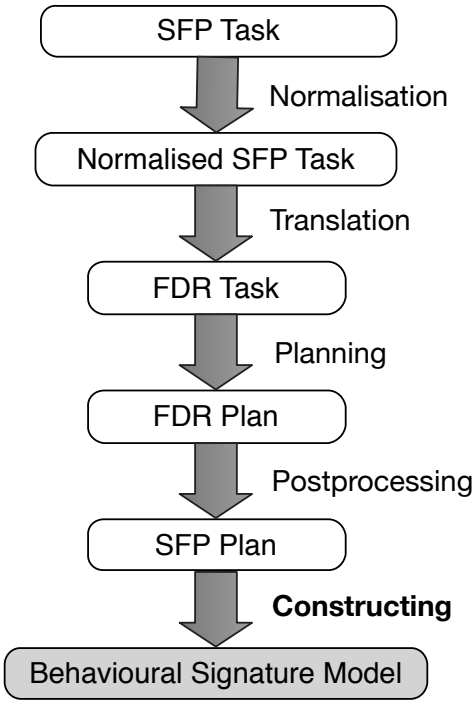


Figure 5.3: Overview of choreographing steps for constructing the Behavioural Signature model based on given SFP configuration task.

5.2.2 Choreographing Behavioural Signature Model

Choreographing aims to automatically constructs a set of BSig models that enables the agents to execute the plan for configuration changes in a distributed way while preserving the ordering constraints. An ordering constraint itself is a constraint where one action must be execution before or after another one in order to satisfy particular conditions e.g. a database service must be started before a web service is started. Failing to satisfy the ordering constraint can make the system entering a period where it does not work properly. For example, a web service fails to generate a correct response since it cannot the database which has not been started yet.

Figure 5.3 gives an overview of the choreographing steps. The first four (normalisation, translation, planning, and postprocessing) are the steps to generate an SFP plan for an SFP configuration task. While the last one is the step to construct the local BSig models from the plan. Since the first four have been described in §4.2, then here we only give details of the last step. We start by giving the definition of the BSig model.

Definition 5.1 (Behavioural Signature Model).
Assume $\mathcal{A} = \{\alpha_1, ..., \alpha_n\}$ is the set of agents, then the *local Behavioural Signature*

model of agent α_i is $B_i = \langle \kappa_i, \hat{A}_i, prov_i, g_i \rangle$, where:

- $\kappa_i \in \mathbb{N}^+$, is the model's serial number;
- $\hat{A}_i = \{ \hat{a}_{ij} \mid \hat{a}_{ij}$ is an SFP grounded action $\}$, is the set of local actions;
- $prov_i : \hat{A}_i \times \hat{V} \rightarrow \mathcal{A}$, is the remote precondition provider function which returns an agent that should be contacted to request the remote precondition.
- g_i, is the partial variable assigment over $\hat{V}_i \cup \hat{V}_g$ called as the local goal.

A *global Behavioural Signature* model $\mathcal{M} = \langle \mathcal{A}, \mathcal{B} \rangle$, where $\mathcal{B} = \{B_1, ..., B_n\}$, and $\forall i, j \,.\, \kappa_i = \kappa_j$. ■

The above definition shows that every local model has the serial number (κ). It must be sent in the peer-to-peer communications to ensure that every agent is executing the same global plan. The agent must decline any request from other agents when it receives a different serial number. Every action ($\hat{a}_{ij}$) in the model is an SFP grounded action whose preconditions and effects are conjunctions over SFP atoms. Since the action precondition can be provided by any agent, then we need to assign a provider agent to every precondition ($prov_i$) so that the agent knows to whom it should send the request of precondition, which can be other agent (*remote precondition*) or itself (*local precondition*). The local goal (g_i) is the partial assignment over local and global variables.

Before constructing the BSig models, an SFP plan π_{SFP} must be generated by the choreographer. This is done centrally where the choreographer aggregates the current state of every agent to get the global knowledge of the system[3]. And then it creates and solves an SFP configuration task Σ using the technique described in §4.2.

There are three things that must be done to construct the BSig model from given SFP plan. First is adding the necessary extra preconditions to particular actions in order to maintain the ordering constraints in regression execution. Second is assigning agents that can provide the preconditions of every action. And third is assigning a set of local goal to every agent.

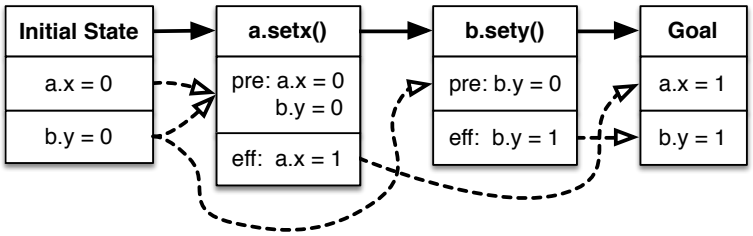
Extra Preconditions

Our regression execution algorithm, which will be introduced later, is purely based on causality: an action will be selected and executed if it can repair the flaws on the goal or the preconditions. However, this selection criteria might violate the global plan's

[3]This is similar with planning step of orchestration technique.

ordering constraints because the constraints are not only based on the causal-links[4], but also the threat resolutions which is shown in algorithm 4.2[5].

For the example, consider the following plan for constructing the BSig models for agent `a` and `b` where arrows are the ordering constraints and dashed-arrows are the causal-links:



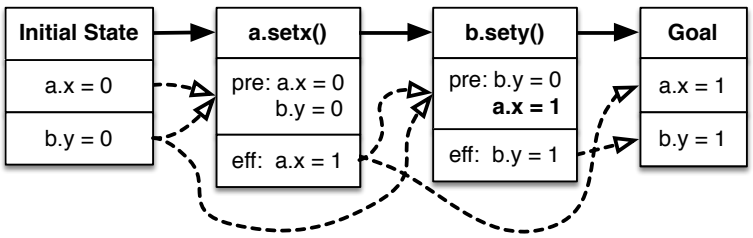
If we do not add extra precondition to replace the threat resolution ordering constraint, in this case `a.setx()` $\prec$ `b.sety()`, then there are two possible execution sequences i.e.:

1. Action sequence: `a.setx()` $\rightarrow$ `b.sety()`
   State transitions: {`a.x=0,b.y=0`} $\rightarrow$ {`a.x=1,b.y=0`} $\rightarrow$ {`a.x=1,b.y =1`}

2. Action sequence: `b.sety()` $\rightarrow$ **failed**
   State transitions: {`a.x=0,b.y=0`} $\rightarrow$ {`a.x=0,b.y=1`} $\rightarrow$ **failed**

The first sequence can reach the goal, while the second cannot. Note that `b.sety()` can be selected in the second sequence since its effect supports the goal `b.y=1` and the initial state `b.y=0` satisfies its precondition.

This problem can be addressed by adding the predecessor's effects to the action's preconditions iff there is such threat resolution ordering constraints between them. Consider the following modified plan:



[4] A causal link is a directed arrow from action a_1 to a_2 where a_1's effects provide a_2's preconditions.

[5] Line 9 adds an ordering constraint based on the causal-link, while line 13 adds the constraint based on the threat resolution.

There is an additional atom `a.x=1` (**bold** text), which is the effect of `a.setx()`, that is added into preconditions of `b.sety()` since its effect (i.e. `b.y=1`) threats a precondition of `a.setx()` (i.e. `b.y=0`). With this extra precondition, the second sequence will never occur since the "new preconditions" of `b.sety()` do not satisfy by the initial state. Thus, only the first sequence which is possible to be executed in regression.

To apply this, we modify the algorithm 4.2 that converts a total-order into partial-order plan. This modification is shown in algorithm 5.1 that has a few extra statements i.e. lines 14-16. These lines will add the predecessor's effect into the current action's preconditions when the action's effects threats the predecessor's preconditions. However, if the variable already exists in the preconditions, then it should not be added since it is an adversary effect. This small modification still maintains the original's complexity which is polynomial if it uses greedy strategy.

Precondition Provider

Finding the agent provider of a precondition is based on the causal link. The agent is set as the provider if it has an action whose effects provide the precondition. Let $(\hat{v} = \upsilon) \in \mathit{Preconditions}(\hat{a}_i)$ is the precondition of action $\hat{a}_i$ of agent i. Then $\mathit{prov}_i(\hat{a}_i, \hat{v}) = id_j$ iff $(\hat{v} = \upsilon) \in \mathit{Effects}(\hat{a}_j)$ where $\hat{a}_j$ and id_j are the action and identifier of agent j respectively.

Local Goal

Every goal in the SFP configuration task must be assigned to an agent whose action provides the goal. This is similar to finding the precondition provider i.e. finding an agent whose action has an effect that provides the goal. Let $\hat{v} = \upsilon$ be the goal, then it is assigned to agent i i.e. $g_i(\hat{v}) = \upsilon$ iff $(\hat{v} = \upsilon) \in \mathit{Effects}(\hat{a})$ and $\hat{a} \in \hat{A}_i$.

Choreographing Algorithm

Algorithm 5.2 summarises the choreographing process. Line 2 is the statement that calls a function that solves an SFP task to generate the global plan. Note algorithm 5.1 (instead of 4.2) must be used in postprocessing in order to add the extra preconditions. Line 3 defines the BSig model's identifier which is assigned to every local model. Lines 4-9 construct the set of local models based on the generated plan. Lines 5-9 construct the local model for agent α_i that identifier id_i. This step finds a set of agents (lines 5-7) that provides the precondition of every local action. Then it assigns a set local

Algorithm 5.1 Generating partial-order plan and add necessary preconditions.

Require: A valid total-order plan $\pi = \langle a^1_\beta, a_1, a^2_\beta, a_2, ..., a^n_\beta, a_n, a^{n+1}_\beta \rangle$ generated after compilation, where a^i_β is the artificial action.

1: **function** GENERATE-PARTIAL-ORDER (π)
2: **for each** action $a_i \in \pi$ **do**
3: $Preconditions(a_i) \leftarrow Preconditions(a_i) \wedge Preconditions(a^i_\beta)$
4: **for** $i \leftarrow n$ down-to 1 **do**
5: **for each** $(v = d) \in Preconditions(a_i)$ **do**
6: **choose** $k < i$ such that:
7: 1. $(v = d) \in Effects(a_k)$, and
8: 2. $\nexists l : k < l < i \,.\, (v = d') \in Effects(a_l) \wedge d \neq d'$
9: add order $a_k \prec a_i$
10: **for each** $(v = d) \in Effects(a_i)$ **do**
11: **for** $j \leftarrow (i-1)$ down-to 1 **do**
12: **if** $(v = d') \in Preconditions(a_j) \wedge d \neq d'$ **then**
13: add order $a_j \prec a_i$
14: **for each** $(v' = d'') \in Effects(a_j)$ **do**
15: **if** $v' \notin Preconditions(a_i)$ **then**
16: $Preconditions(a_i) \leftarrow Preconditions(a_i) \wedge (v' = d'')$ ▷ Add preconditions.
17: **return** $\langle \{a_1, a_2, ..., a_n\}, \prec \rangle$

Algorithm 5.2 Choreographing the BSig models for an SFP configuration task.

Require: Σ is the SFP task and $agents = \{\alpha_1, ..., \alpha_n\}$ is the set of agents.

1: **function** CHOREOGRAPHING($\Sigma, agents$)
2: $\pi_{SFP} \leftarrow$ SOLVE-SFP-TASK(Σ) ▷ see §4.1
3: $\kappa \leftarrow$ any unique number ▷ BSig model's identifier
4: **for** $i \leftarrow 1$ to n **do**
5: **for each** action $\hat{a}_i \in \pi_{SFP}$ where $Name(\hat{a}_i) = id_i :: r$ **do** ▷ id_i is the identifier of agent α_i
6: **for each** $(\hat{v} = \upsilon) \in Preconditions(\hat{a}_i)$ **do** ▷ precondition provider
7: $prov_i(\hat{a}_i, \hat{v}) = \alpha_j$ iff $\exists \hat{a}_j \,.\, (\hat{v} = \upsilon) \in Effects(\hat{a}_j)$
8: **for each** goal $(\hat{v} = \upsilon)$ **do** ▷ the local goal of agent α_i
9: $g_i(\hat{v}) = \upsilon$ iff $(\hat{v} = \upsilon) \in Effects(\hat{a}_i)$
10: **return** $\langle agents, \{B_i\}_{i=1}^{n} \rangle$

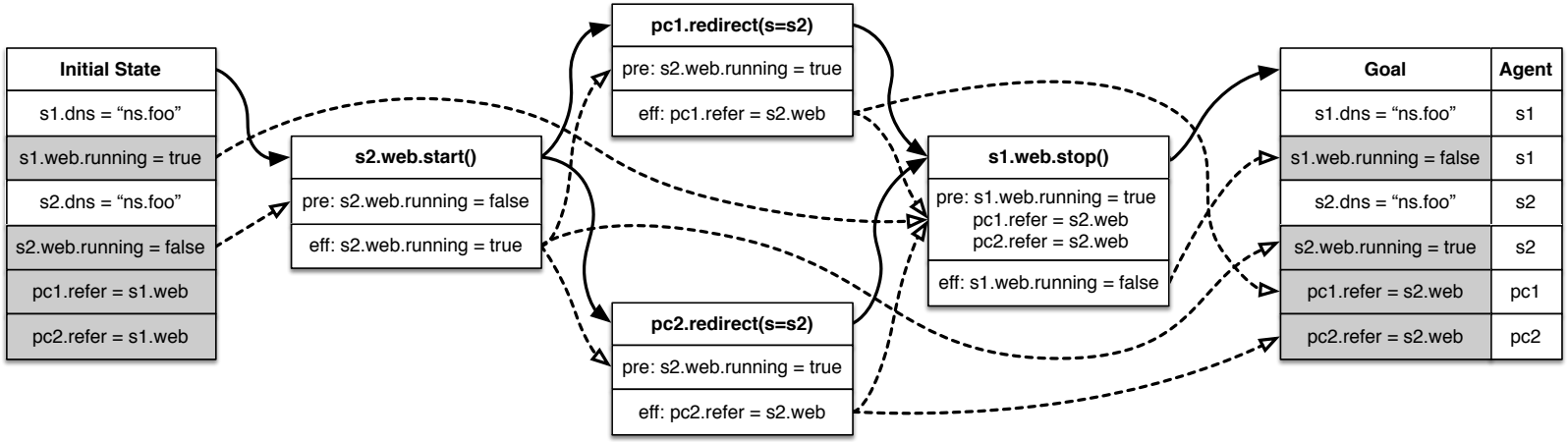


Figure 5.4: The global plan for choreographing the BSig models for the multi-services system (see figure 3.1), where the current and the desired state are given in listing 3.10 and 3.10 respectively. Arrows are the ordering constraints. The precondition providers and the local goals are generated based on the causal-links (dash-arrows). Every goal is assigned to the right side agent.

goal (lines 8-9) to the agent. Finally, it returns the global BSig model which is ready to be deployed to the target agents. The complexity of the algorithm is dominated by the planning step since the model construction's complexity is linear to the number of actions of the plan.

Example 1: Multi-Services System

Figure 5.4 illustrates the choreographing process for a configuration task of the multi-services system described in §3.3. The SFP plan consists of four actions that change the state variables' (grey) value from initial to the goal. The arrows are the ordering constraints generated by the plan which achieves the goal and preserves global constraints. The dashed-arrows show the causal links of the goal and precondition which are used to determine the agent providers and the local goals.

The following YAML codes are the global BSig model for this task generated by the choreographer. There are four local models i.e. lines 1-16, lines 17-26, lines 27-36, and lines 37-46, which are deployed to agent `s1`, `s2`, `pc1` and `pc2` respectively. As you may notice that every local model has the same identifier: `10001`. The local goal (`goals`) of each agent is a set of variable-value pairs. Each model has a set of local actions (`actions`), each of which has a name, a set of parameters, conditions (conjunction of SFP atoms), and effects (conjunction of SFP atoms). The model also has a list of precondition providers (`provider`).

```
1  s1:
2    id:    10001
3    goals: {s1.dns: "ns.foo", s1.web.running: false}
4    actions:
5      - name:       s1.web.stop
6        parameters: {}
7        condition:
8          s1.web.running: true
9          pc1.refer:      s2.web
10         pc2.refer:      s2.web
11       effect:     {s1.web.running: false}
12   provider:
13     s1.web.stop:
14       s1.web.running: s1
15       pc1.refer: pc1
16       pc2.refer: pc2
17 s2:
18   id: 10001
19   goals: {s2.dns: "ns.foo", s2.web.running: true}
20   actions:
21     - name:       s2.web.start
22       parameters: {}
23       condition:  {s2.web.running: false}
```

```
24        effect:     {s2.web.running: true}
25    provider:
26      s2.web.start: {s2.web.running: s2}
27  pc1:
28    id: 10001
29    goals: {pc1.refer: s2.web}
30    actions:
31      - name:       pc1.redirect
32        parameters: {s: s2}
33        condition:  {s2.web.running: true}
34        effect:     {pc1.refer: s2.web}
35    provider:
36      pc1.redirect: {s2.web.running: s2}
37  pc2:
38    id: 10001
39    goals: {pc2.refer: s2.web}
40    actions:
41      - name:       pc2.redirect
42        parameters: {s: s2}
43        condition:  {s2.web.running: true}
44        effect:     {pc2.refer: s2.web}
45    provider:
46      pc2.redirect: {s2.web.running: s2}
```

Example 2: Simple Cloud System

Consider another system described in §5.2.1. This simple cloud system is different than the previous one because it has not only local (`cloud1.running` and `cloud2.running`) but also a global variable (`vm1.in_cloud`). The system has three agents i.e. `cloud1`, `cloud2` and `vm1`. The first two agents have an action that can change the value of `vm1.in_cloud`. Assume we define the following desired state for this system.

```
1  include "schemata.sfp"
2  // desired state
3  main {
4    cloud1 isa Cloud { running  = true;    }
5    cloud2 isa Cloud { running  = true;    }
6    vm1    isa VM    { in_cloud = cloud2; }
7  }
```

The above specification shows that variable `vm1.in_cloud` is going to be changed from `cloud1` to `cloud2`. In other words, `vm1` will be migrated from one to another cloud infrastructure. Figure 5.5 illustrates the choreographing that implements such configuration task. It shows that the goal `vm1.in_cloud=cloud2` must be assigned to agent `cloud2` whose action `cloud2.create_vm` provides it. While agent `cloud1` must be set as the provider of the precondition `vm1.in_cloud=null` of action `cloud2.create_vm`. The following YAML codes show the global BSig model for this task. It consists of

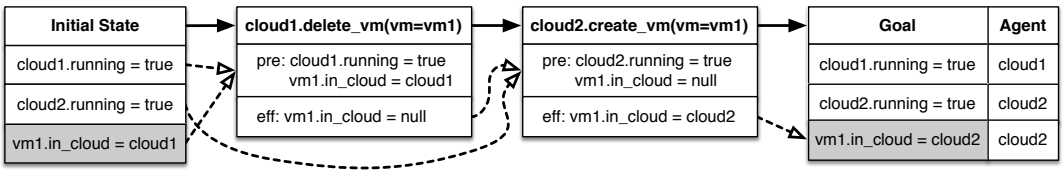


Figure 5.5: The global plan for choreographing the BSig models of simple cloud system. The precondition providers and the local goals are generated based on the causal-links (dash-arrows). Every goal is assigned to the right side agent.

three blocks i.e. lines 1-12, 13-24, and 25-29, which are deployed to agent `cloud1`, `cloud2` and `vm1` respectively.

```
1  cloud1:
2    id: 10002
3    goals: {cloud1.running: true}
4    actions:
5      - name:       cloud1.delete_vm
6        parameters: {vm: vm1}
7        condition:  {cloud1.running: true, vm1.in_cloud: cloud1}
8        effect:     {vm1.in_cloud: cloud1}
9    provider:
10     cloud1.delete_vm:
11       cloud1.running: cloud1
12       vm1.in_cloud:   vm1
13 cloud2:
14   id: 10002
15   goals:   {cloud2.running: true, vm1.in_cloud: cloud2}
16   actions:
17     - name:       cloud2.create_vm
18       parameters: {vm: vm1}
19       condition:  {cloud2.running: true, vm1.in_cloud: null}
20       effect:     {vm1.in_cloud = cloud2}
21   provider:
22     cloud2.create_vm:
23       cloud2.running: cloud2
24       vm1.in_cloud:   cloud1
25 vm1:
26   id: 10002
27   goals:    {}
28   actions:  []
29   provider: {}
```

5.2.3 Executing Behavioural Signature Model

A global BSig model $\mathcal{M}$ is deployed by sending each local model $B_i \in \mathcal{M}$ to agent $\alpha_i \in \mathcal{A}$. This can be done asynchronously i.e. whenever an agent receives the model, it immediately stops all processes and then replace the existing model with the new one and resume the execution. The agent does not need to wait until all local models have been deployed since the serial number (κ) can be used to check whether the agents' local model are part of the same global model or not.

Every local model is executed using the *cooperative regression reactive* (CRR) algorithm, which is motivated by two objectives. First is synchronising the agents' executions to preserve the global plan's ordering constraints. Second is minimising the requirement of re-choreographing (replanning).

The first objective obligates the agents to be *cooperative* in order to maintain the ordering constraint across the agents. However, the synchronisation is not achieved using a time-based, but a *state-request-response* approach. The latter is better because it does not require the agents to have the same time clock, which is usually very hard to be implemented in heterogeneous environment. In addition, time-based is not reliable since the execution time of the same action by the same agent may differ from one to another occasion due to unpredictable load of resources such as CPU. On the other hand, state-request-response approach only requires the agent sending the goal that should be achieved by other agents and then acting based on their response. Although it does not depend on time, but a communication link must be established between the agents. In practice, this could be a problem in particular when there is no direct communication link between them, for example an agent must communicate with another agent which is behind a firewall. A solution for this problem might be using a proxy as a bridge to enable "indirect" communication.

The second objective is achieved by executing the plan using *regression* technique. It is inspired by the fact that if the plan is executed in progression then it is only valid when the system is at the initial state of the plan. However, if it is executed in regression then it is valid when the system is either at the initial or one of the intermediate states of the plan. This could avoid unnecessary re-choreographing, in particular when a reconfiguration is needed to correct any drift from the desired state. Hence, the reconfiguration cost can be reduced since re-choreographing requires the choreographer agent to aggregating the current state of every agent (communication cost) and planning (computation cost). In their works, [Fritz and McIlraith, 2007, Muise et al., 2011]

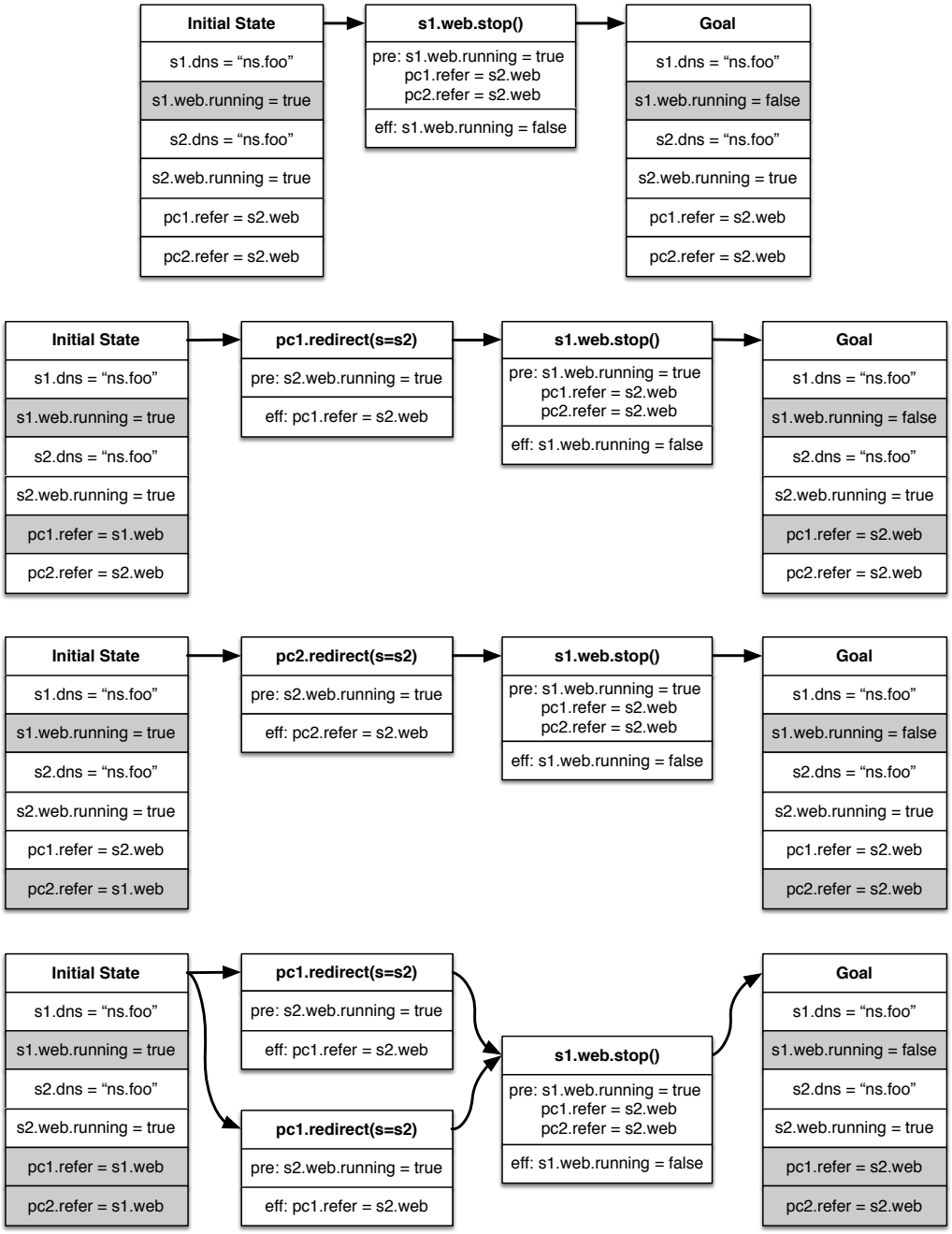


Figure 5.6: These are the other four possible states, besides the initial state, when executing the plan in figure 5.4 with regression.

have shown that the regression is a viable technique to execute the plan in single agent setting and proved to be robust on environment changes. However, we are not aware of any work that uses regression for executing a plan in multi-agents setting.

The advantage of regression comparing to progression is described by example. Consider the plan in figure 5.4 which is the solution for the configuration task of the

multi-services system described in §3.3. With progression, the plan is only valid when the system is at the initial state i.e.:

{ `s1.dns="ns.foo"`, `s1.web.running=true`, `s2.dns="ns.foo"`,
`s2.web.running=false`, `pc1.refer=s1.web`, `pc2.refer=s1.web` }

But with regression, the plan is valid for five possible states (including the initial state) where the other four are illustrated in figure 5.6. This covers 5 times more possible current states than progression.

This is possible because our algorithm will only select and execute an action that can repair the existing goal flaws. The selection is purely based on the causal-links and formally defined as follows:

Definition 5.2 (Action Selection). Assume $(\hat{v} = \upsilon) \in$ the current state and $(\hat{v} = \upsilon') \in$ the goal, then $(\hat{v} = \upsilon')$ is a flaw iff $\upsilon' \neq \upsilon$. Given a set of flaws $\{(\hat{v}_1 = \upsilon'_1), ..., (\hat{v}_n = \upsilon'_n)\}$, then an action $\hat{a} \in \hat{A}$ can be executed ($\hat{a}$ is applicable) iff $\exists i \,.\, (\hat{v}_i = \upsilon'_i) \in \mathit{Effects}(\hat{a})$. ■

The above definition shows that the selection is non-deterministic: the agent can execute any applicable action. However, in practice, we might want to have a deterministic selection due to some reasons, for example: a debugging process. This can be easily done by sorting the actions into a total-order (a list), and then select the first or the last applicable action in the list.

At this point, you may notice that this thesis has used several representations:

- SFP (introduced in §3) is a *high-level* representation for system administrator to specify a configuration task;
- FDR (§2.2.3) is a *low-level* representation as the input for a classical planner to solve a planning problem, and every SFP task can be automatically compiled into an FDR task using a technique described in §4.2;
- YAML (§5.2.2) is a *low-level* representation to specify a BSig model, which is automatically generated using a technique described in §5.2.2.

Every BSig model provides useful information to an agent on: 1) how to rationally select an action, and 2) what preconditions that must be satisfied before execution and who (agents) that should be contacted. However, it does not specify "when" the agent should communicate with the others and what are the "constraints" of this communication process[6]. A concise specification of agents communication is critical since it

[6]SFP and FDR cannot be used as well because they do not have the notion of communication.

```
1  a(single(K,A,Prov,G),PID)::
2    null <- getFlaws(G,Flaws) then
3    null <- isEmpty(Flaws) or
4    (
5      null <- getAction(Flaws,A,Act) then
6      (
7        (
8          null <- getPreconditions(Act,Pre) then
9          a(local(K,A,Prov,Pre),PID) then
10         null <- execute(Act)
11       )
12     ) then
13     a(single(K,A,Prov,G),PID)
14   ).
```

Figure 5.7: The LCC interaction model of BSig execution for single agent system.

can affect the execution result e.g. ensuring that the global constraints are preserved during execution.

On the other hand, Lightweight Coordination Calculus (LCC) [Robertson, 2005] is a language based on a process algebra that has notations to concisely specify communication protocols between agents. Since the execution of BSig models require the communications between agents, then the remaining of this subsection will use LCC to specify the regression algorithm implemented in the agents.

Regression Algorithm for Single Agent

Before describing the multiagent algorithm, first we describe the regression algorithm for a single agent which is shown in figure 5.7. The algorithm is written as interaction model (IM) *single* in Lightweight Coordination Calculus (LCC). This is because, later in multiagent algorithm, we want to describe the communications (interactions) between the agents. Although there will be no interaction in this single agent algorithm, but for the sake of consistency, it is also defined in LCC. Our algorithm is similar to the one introduced in [Muise et al., 2011], but they are different on the method for selecting the applicable action: ours is based on the causal-links only, while the other is based on the partial ordering constraints.

Assume $B = \langle \kappa, \hat{A}, prov, g \rangle$ is the local BSig model which is going to be executed. Since this only involves one agent, then the algorithm does not use *prov* because the provider of every precondition is always the agent itself. The execution can be started by passing the model's elements as the following:

```
a(single(K, A, Prov, G), agent).
```

where: K $= \kappa$, A $= \hat{A}$, Prov $= prov$, and G $= g$. The details of the algorithm can be described as follows:

Line 1 K is the model's serial number (κ), A is the set of local actions (A_i), Prov is the agent precondition provider function (*prov*), G is the local goal (g_i), and PID is the process ID which can be set with agent's ID.

Line 2 Constraint getFlaws will perform sensing to get the current state and then compare it with the goal to compute and return the flaws (Flaws) that should be repaired. This constraint always returns true.

Line 3 Constraint isEmpty returns true if the flaws is empty and then the execution will finish **successfully**. Otherwise it returns false which triggers the execution of lines 5-13.

Line 5 Constraint getAction is finding an action in A that can repair the flaws (see definition 5.2). If the action is found, then it sets the action to variable Act and then returns true. Otherwise, it returns false which will end the execution with **failure**.

Line 8 Constraint getPreconditions will set the precondition of the selected action Act to variable Pre. It always returns true.

Line 9 In order to satisfy the preconditions, a recursive call is made by passing Pre as the new goal, while others (K, Prov, and A) are the same. Whenever the recursive call is finish successfully then it continues to the next statement – this means that Pre has been achieved. But, if it is failed, then the entire execution will end with **failure**.

Line 10 Since the precondition has been achieved then action Act can be executed. Constraint execute returns true if the execution was success, otherwise it returns false which ends the entire execution with **failure**.

Line 13 A recursive call is made using the same goal to ensure that every goal has been achieved because the previous action execution may only achieve the sub-goal. If there is no flaw on the goal then the execution will end **successfully**.

The regression execution can be applied to the orchestration simply by implementing this single agent algorithm into the controller agent.

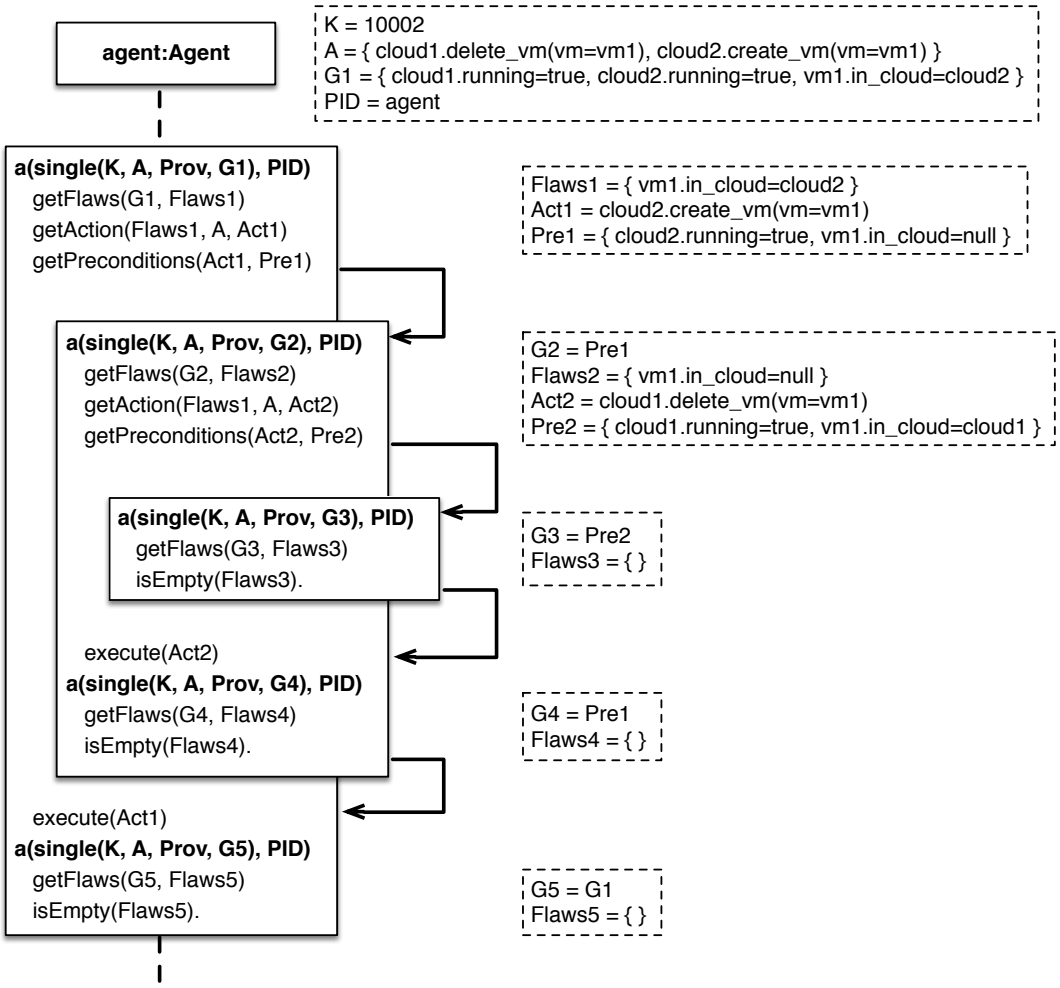


Figure 5.8: The sequence diagram of single agent BSig execution for the simple cloud system. G_i and Flaw$_i$ are the goal and flaws that should be achieved and repaired by the agent. Act$_i$ is the action that will be executed, and Pre$_i$ is the precondition of Act$_i$ that should be satisfied before execution.

To show how the algorithm works, consider the simple cloud system described in §5.2.1. The global plan for this system is shown in figure 5.5. Assume that all resources are managed by a single agent. Hence, all actions and goals are in the local BSig model of this agent.

The sequence of regression execution is illustrated in figure 5.8. It begins when the model is passed to interaction model *single*. Based on the current state, a flaw `vm1.in_cloud=cloud2` (Flaws1) is found on the goal (G1). To repair the flaw, it then performs two recursive calls. The first one is achieving the precondition `vm1.in_cloud=null` (Pre1) of action `cloud2.create_vm(vm=vm1)` (Act1) that can repair the goal flaw

(Flaws1). The second recursive is ensuring that the precondition (Pre2) of action `cloud1.delete_vm(vm=vm1)` (Act2) is satisfied by the current state, which is then executed to provide the precondition of `cloud2.create_vm(vm=vm1)` (Act1) which is executed afterwards. Hence, the goal flaw has been repaired and the whole execution is finish successfully.

Regression Algorithm for Multi-Agents

The multi-agents regression algorithm extends the single agent version in three ways. First, before executing an action, it classifies the preconditions into local or remote: the local preconditions are achieved in similar way as the single agent algorithm that is through a recursive call by passing the preconditions as the new goal; and the remote preconditions are sent to the provider agents as specified in the local BSig model. Second, every agent has a process that receives goal requests from other agents. The requested goal is then achieved by finding its flaws and then executing the necessary local actions to repair them. It sends an "ok" response if there is no flaw on the requested goal, or when the flaw cannot be repaired then it will stop all processes and trigger a re-choreographing by sending a request to the choreographer agent. Third, it defines the internal process that periodically achieves the local goal in order to enable the self-healing capability.

Figure 5.9 shows two interaction models i.e. *local* (lines 1-20) and *remote* (lines 22-32) that implement the first extension. Similar with *single*, IM *local* tries to achieve given goal. On the other hand, IM *remote* requests the remote preconditions to all provider agents concurrently. These two IMs can be described in details as follows:

Lines 1-5 These are equivalent with IM *single* where the agent computes the goal flaws. If such flaws exist, then it selects an action (`Act`) that can repair them.

Lines 6-9 Constraint `lock` (line 7) is trying to lock the selected action (`Act`) before execution. It is required to ensure an atomic action execution since there might be multiple requests from different agents that can be achieved by the same action[7](the state of particular resource cannot be changed by two executions in the same time). The constraint returns false if the action is already locked and then the process will have to wait (line 8) for some times. It will restart the process by re-invoking the same IM to achieve the same goal (line 19). If it can lock, then the execution continues to lines 11-17.

[7]This behaviour can be implemented using a set of IMs, each of which is handling an action, and the

```
1  a(local(K,A,Prov,G),PID)::
2    null <- getFlaws(G,Flaws) then
3    null <- isEmpty(Flaws) or
4    (
5      null <- getAction(Flaws,A,Act) then
6      ( // try to lock action Act
7        null <- not lock(Act) then
8        null <- wait()
9      )
10     or
11     ( // LG: local preconditions, RG: remote preconditions
12       // RA: list of agents that provide RG
13       null <- classifyPreconditions(Act,Prov,LG,RG,RA) then
14       a(local(K,A,Prov,LG),PID) then
15       a(remote(K,RG,RA),PID) then
16       null <- execute(Act) and unlock(Act)
17     )
18     then
19     a(local(K,A,Prov,G),PID)
20   ).
21
22 a(remote(K,Goals,Agents),PID)::
23   null <- isEmpty(Goals) or
24   (
25     null <- Agents = [AH|AT] and Goals = [GH|GT] then
26     (
27       goal(GH) => a(satisfier(K,_,_),AH) then
28       equals(S,ok) <- status(S) <= a(satisfier(K,_,_),AH)
29     )
30     par
31     a(remote(K,GT,AT),PID)
32   ).
```

Figure 5.9: The *local* IM that finds and executes the necessary actions that can repair the flaws of given goal. It calls itself recursively to achieve the local precondition, and starts *remote* IM to send the remote preconditions to other agents concurrently.

Line 13 Constraint `classifyPreconditions` classifies the preconditions of action `Act` into local `LG` and remote `RG` based on agent providers `Prov`. Note that `RA` contains the list of agents that provide the remote preconditions.

Line 14 The local preconditions are achieved through recursive call by passing `LG` to IM *local* as the new goal while other parameters (`K`, `A`, and `Prov`) are the same. When the recursive call ends with failure, it stops all processes and triggers re-choreographing. Otherwise, it continues to the next line.

Line 15 The remote preconditions are achieve by invoking IM *remote* and passing `K`,

lock is obtained by sending a message to this action IM.

`RG` and `RA` as arguments. When this invocation ends successfully, it means every remote preconditions have been achieved. Thus, it will continue to the next line because. Otherwise, the execution ends with failure.

Line 16 Since local and remote preconditions have been achieved, the selected action `Act` is then executed by calling constraint `execute`. If the execution is success, the action will be unlocked. Then it invokes the same IM (line 19) with the same goals in order to ensure that every goal has been achieved.

Line 22-32 These are IM *remote* that sends requests of remote preconditions to target agents concurrently. It splits the goal (line 25) for an agent, and then sends the goal request (line 27). Whenever a response is received (line 28), it checks whether the response is "ok" which means the request has been fulfilled. Otherwise, it ends the execution with failure.

Figure 5.10 shows three IMs: *satisfier* (lines 16-29) that listens for any goal request from other agent (the second extension); *healer* (lines 5-14) that periodically achieves the local goal for self-healing (the third extension); and *multi* (lines 1-3) which is the main IM.

IM *multi* is invoked by the agent after receiving the local BSig model from the choreographer. Assume *id* is the agent's identifier and $B = \langle \kappa, \hat{A}, prov, g \rangle$ is the local model, then similar to the single agent algorithm, it starts the model execution by passing every element as the follows:

```
a(multi(K, A, Prov, G), PID).
```

where: `K` $= \kappa$, `A` $= \hat{A}$, `Prov` $= prov$, `G` $= g$, and `PID` $= id$. The main process will be split-up into two processes, one is invoking *satisfier* (line 2) and another is invoking *healer* (line 3).

IM *satisfier* is invoked by passing the model's serial number (`K`), the local actions (`A`), and the agent providers (`Prov`). It then waits until it receives a message of goal request (line 17) from other agents. It tries to achieve the goal by invoking IM *local* (line 19). If it ends successfully, which means that the requested goal has been achieved, then it sends an "ok" response (line 20) to the requester agent. However, if it is failed, it sends a "failed" response (line 23), stops all processes (line 24), and triggers re-choreographing. Note that after receiving a message, it spawns another thread (lines 28-29) which duplicates itself in order to avoid blocking so that other requests can be served concurrently.

```
1  a(multi(K,A,Prov,G),PID)::
2    a(satisfier(K,A,Prov),PID) par // listen for goal request
3    a(healer(K,A,Prov,G),PID).     // start self-healing
4
5  a(healer(K,A,Prov,G),PID)::
6    (
7      ( // try to achieve local goal
8        a(local(K,A,Prov,G),PID) then
9        null <- sleep() // success; sleep until next cycle
10     ) or
11     null <- stop()    // failure stop all processes, and
12                       // then trigger re-choreographing
13   ) then
14   a(healer(K,A,Prov,G),PID). // start next cycle
15
16 a(satisfier(K,A,Prov),PID1)::
17   goal(RG) <= a(remote(K,_,_),PID2) then
18   ( // try to achieve requested goal (RG)
19     a(local(K,A,Prov,RG),PID1) then
20     status(ok) => a(remote(K,_,_),PID2) // success
21     or
22     ( // failure
23       status(failed) => a(remote(K,_,_),PID2) then
24       null <- stop() // stop all processes, and then
25                      // trigger rechoreographing
26     )
27   )
28   par
29   a(satisfier(K,A,Prov),PID1). // listen another request
```

Figure 5.10: *Multi* is the main interaction model, *satisfier* is the interaction model for receiving and achieving any goal request from other agent, and *healer* is the interaction model for achieving the local goal (self-healing).

On the other hand, IM *healer* is invoked by passing every model's element: the model's serial number (`K`), the local actions (`A`), the agent providers (`Prov`), and the local goals (`G`). It starts by trying to achieve the local goal by invoking IM *local* (line 8). If it ends successfully, it sleeps (line 9) until the next cycle (line 14). On the other hand, if it fails[8], then it stops all processes and triggers re-choreographing (line 11).

These five IMs are parts of the multi-agents execution algorithm. Figure 5.11 illustrates the agent transitions between these IMs during execution.

[8]Whenever the agent fails to achieve a local or requested goal, this means that the model is invalid due to some reasons e.g. no action can provide the goal. Thus, the execution process must be stopped so that a new model can be re-choreographed.

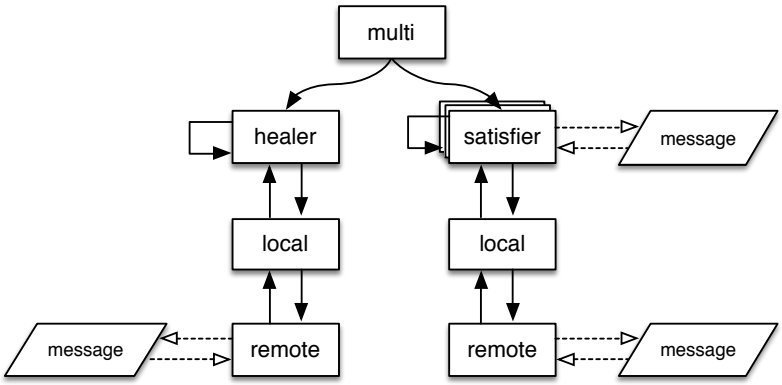


Figure 5.11: The agent transitions between interaction model *multi*, *healer*, *satisfier*, *local*, and *remote* when it executes the multi-agents regression algorithm.

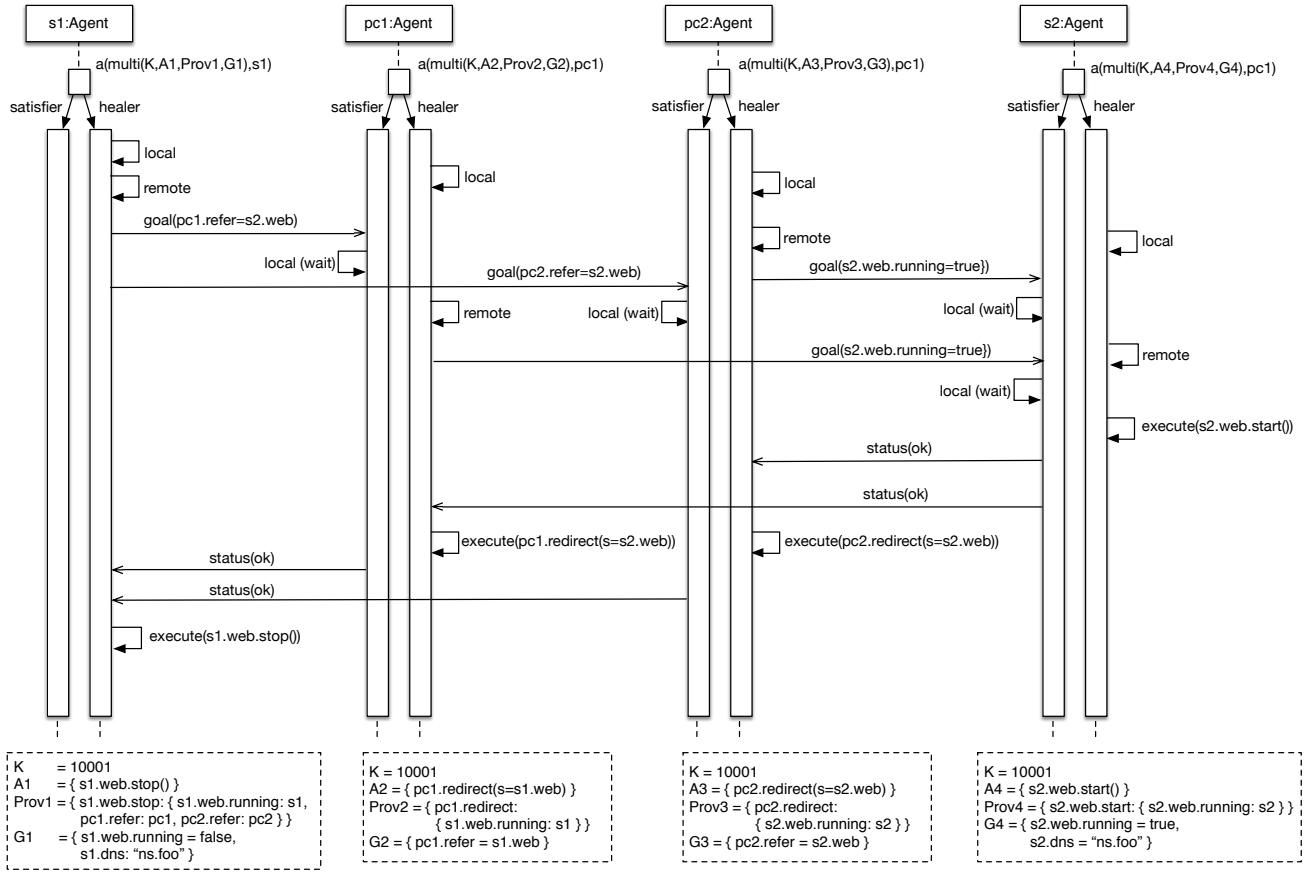


Figure 5.12: Example: multi-services system – the interaction diagram of agents when executing the BSig models based on the plan depicted in figure 5.4.

Example 1: Multi-Services System

Consider the example multi-services system in §3.3 whose global plan is depicted in figure 5.4. Figure 5.12 illustrates the interactions between the agents during the execution of the BSig models. Every agent starts the execution by invoking IM *multi*. It then spawns two processes, one is invoking IM *satisfier* and another is invoking IM *healer*.

For agent `s1`, its *healer* process tries to repair the flaw of its local goal i.e. `s1.web.running=false`. However, since action `s1.web.stop()` has two remote preconditions i.e. `pc1.refer=s2.web` and `pc2.refer=s2.web`, it sends the goal requests to agent `pc1` and `pc2` respectively.

For agent `pc1`, its *healer* process wants to repair the flaw of its local goal i.e. `pc1.refer=s2.web` by selecting and locking action `pc1.redirect(s=s2.web)`. But before execution, it invokes IM *remote* to request a remote precondition `s2.web.running=true` to agent `s2`, and then wait its response. On the other side, its *satisfier* process receives a requested goal `pc1.refer=s2.web` from `s1`. However, this process must wait since action `pc1.redirect(s=s2.web)` is being locked by *healer* process.

A similar situation is happening in agent `pc2` where its *healer* process requests a remote precondition `s2.web.running=true` to agent `s2`, which is required before executing action `pc2.redirect(s=s2.web)`. Its *satisfier* process receives a requested goal `pc2.refer=s2.web` from agent `s1`.

For agent `s2`, its *healer* process tries to repair a local goal flaw `s2.web.running=true` by selecting action `s2.web.start()`. However, before execution, it invokes IM *local* and *remote* to ensure all preconditions have been satisfied. Afterwards, it executes the action. On the other side, its *satisfier* process receives two same requests (`s2.web.running=true`) from `pc1` and `pc2`. It has to wait since the supporting action is being locked by the *healer* process. When the execution has finished, hence the requested goal has been achieved, it then sends "ok" responses to `pc1` and `pc2`.

Afterwards, agent `pc1` and `pc2` receive an "ok" response from `s2`. And then they continue executing the selected actions i.e. `pc1.redirect(s=s2.web)` and `pc2.redirect(s=s2.web)`, which repair their local goal flaw as well as achieve the requested goals from `s1`. Thus, their *satisfier* process send an "ok" response back to `s1`.

Finally, agent `s1` receives an "ok" response from `pc1` and `pc2`. Since all precon-

ditions have been satisfied, it then continues by executing action `s1.web.stop()` that repair its local goal flaw. This concludes the result where all agents have achieved their local goals.

Example 2: Simple Cloud System

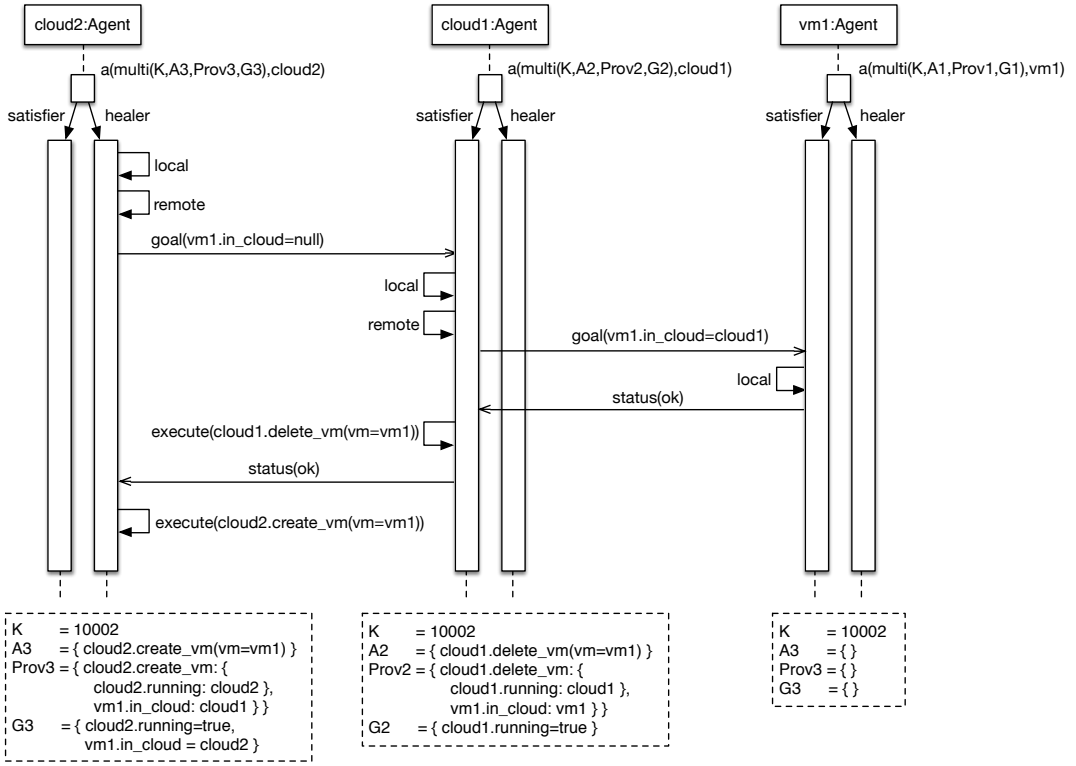


Figure 5.13: Example: simple cloud system – the interaction diagram of agents when executing the BSig models based on the plan depicted in figure 5.5.

Consider the example simple cloud system in §5.2.1 whose global plan is given in figure 5.5. Figure 5.13 shows the interaction diagram of the agents when executing the BSig models. Similar with the previous example, the execution is started when the agents are invoking IM *multi*, each of which then spawns two processes, one is invoking IM *satisfier* and another is invoking IM *healer*.

For agent `cloud2`, it finds local goal flaw `vm1.in_cloud=cloud2` and then selects action `cloud2.create_vm(vm=vm1)` that can repair it. But before execution, it sends the action's remote precondition `vm1.in_cloud=null` to agent `cloud1` as defined in the model's agent providers.

For agent `cloud1`, its *healer* process does not find any flaw on its local goal. But its *satisfier* process receives goal request `vm1.in_cloud=null` from `cloud2`, and then

tries to achieve it by selecting action `cloud1.delete_vm(vm=vm1)`. However, it sends the action's remote precondition `vm1.in_cloud=cloud1` to agent `vm1` in order to ensure that all preconditions have been satisfied.

The *healer* process of agent `vm1` does not have any flaw on its local goal. On the other hand, its *satisfier* process receives goal request `vm1.in_cloud=cloud1` from `cloud1`. Since the current state satisfies the goal request, it sends back "ok" response.

After receiving "ok" response from `vm1`, agent `cloud1` is then executing action `cloud1.delete_vm(vm=vm1)` since all preconditions have been satisfied. Afterwards, it sends an "ok" response back to agent `cloud2`.

Since agent `cloud2` receives "ok" response from `cloud1`, it then continues by executing action `cloud2.create_vm(vm=vm1)`. This repairs its goal flaw and concludes the whole execution.

5.2.4 Correctness

The main advantage of using a planner is that it can generate a plan which not only achieves the goal, but the plan itself is guaranteed to be deadlock and livelock free due to its acyclic ordering constraints. Thus, either the choreographing or the execution algorithm must preserve these constraints so that the same properties can be maintained during execution. We define two correctness properties of our choreography technique. First is the soundness of the action selection in order to ensure that the goal will be achieved. Second is guaranteeing that deadlock or livelock situation will never be occurred.

Theorem 5.3. *The action selection in the multi-agents regression execution algorithm is sound.*

Proof. The action selection is sound since the execution algorithm uses the rule in definition 5.2 to select the action. Based on the causal-links, the rule selects an action only if the action's effects provides the goal, and it never selects an action whose effects do not provide the goal. □

Theorem 5.4. *Assume $\mathcal{M} = \langle \mathcal{A}, \mathcal{B} \rangle$ is the global BSig model generated by the choreographing algorithm 5.2. If $\mathcal{M}$ is executed using the multi-agents regression execution algorithm then the agents will never reach any deadlock or livelock situation.*

Sketch of Proof. The essence of the proof is based on an assumption that the planner generates a partial-order plan $\pi = \langle \hat{A}, \prec \rangle$ where $\prec$ is acyclic. Then, it must be shown

that the execution of the BSig model constructed from π will preserve $\prec$.

The choreographing algorithm 5.2 shows that the partial-order plan π, which is used to construct $\mathcal{M}$, is generated by the modified partial-order generator algorithm 5.1. The algorithms adds $\hat{a}_i \prec \hat{a}_j$ iff the effects of $\hat{a}_i$ supports the preconditions of $\hat{a}_j$ (causal-link), or the effects of $\hat{a}_j$ threats the preconditions of $\hat{a}_i$ (threat-resolution). It also adds the effects of $\hat{a}_i$ into the preconditions of $\hat{a}_j$ (extra preconditions) iff $(\hat{a}_i \prec \hat{a}_j) \in \prec$ and $\hat{a}_j$ threats $\hat{a}_i$.

By contradiction, assume that the execution is not preserving $\prec$ i.e. there is such situation where $(\hat{a}_i \prec \hat{a}_j) \in \prec$ and $\hat{a}_j$ is executed before $\hat{a}_i$. However, this will never be occurred because of two reasons. First, the execution algorithm always finds and repairs every flaw on the preconditions before executing the action by executing another action that can repair the flaws. This behaviour always satisfies the ordering constraint based on the causal-links. Second, since the effects of $\hat{a}_i$ are the preconditions of $\hat{a}_j$, then the effects become the flaws. Thus, the algorithm will always execute $\hat{a}_i$ first since it has to repair the flaws before executing $\hat{a}_j$.

Since it has been shown that such situation will never be occurred then the execution is always preserving $\prec$. Thus, there will be no deadlock or livelock during execution.

□

5.2.5 Progression Execution with Idempotent Actions

Some configuration tools, such as Ansible [Ansible Inc., 2014] and Chef [Opscode Inc., 2014], are relying on a script containing a workflow (plan) with *idempotent* actions to enable the self-healing capability, where the script is executed progressively and periodically. The idempotent action is a concept where the action leaves the system unmodified if it is already at the desired state. This behaviour can be formally described using conditional effects. Assume we have a "normal" action a where:

$$\mathit{Preconditions}(a) = \psi \text{ and } \mathit{Effects}(a) = \phi$$

a can be converted into idempotent action a' such that:

$$\mathit{Preconditions}(a') = \mathit{true} \text{ and } \mathit{Effects}(a') = (\text{when } \psi \wedge \neg\phi \text{ then } \phi)$$

Note that the action is always executable since the preconditions is always true, while the effects are applied when its conditions are satisfied by the current state.

Naively, we could convert the plan's actions into idempotent ones and execute them

in progression, and then expect that the results will be same as our regression technique. However, although both could achieve the desired state, but the state transitions are not the same – the first might employ unnecessary changes, while the latter will not. These unnecessary changes might produce undesired behaviours during execution.

For illustration, consider a simple example of configuration task where we want to upgrade a running service `s`. The task can be described as follows:

initial state:
`s.running=true`
`s.version=1`

goal:
`s.running=true`
`s.version=2`

action: `s.start`
pre: `s.running=false`
eff: `s.running=true`

action: `s.stop`
pre: `s.running=true`
eff: `s.running=false`

action: `s.upgrade`
pre: `s.running=false` $\wedge$ `s.version=1`
eff: `s.version=2`

The solution plan for this task is: `s.stop` $\rightarrow$ `s.upgrade` $\rightarrow$ `s.start`.

The plan's actions could be converted into idempotent ones which can be described as follows:

action: `s.start`
pre:
eff: when (`s.running=false`) then (`s.running=true`)
action: `s.stop`

pre:
eff: when (`s.running=true`) then (`s.running=false`)
action: `s.upgrade`

pre:
eff: when (`s.running=false` $\wedge$ `s.version=1`) then (`s.version=2`)

If the service is at the initial state and the plan with idempotent actions is executed progressively, then we will get state transitions:
`{s.running=true, s.version=1}` $\rightarrow$ `{s.running=false, s.version=1}` $\rightarrow$
`{s.running=false, s.version=2}` $\rightarrow$ `{s.running=true, s.version=2}`
Obviously, this execution can achieve the desired state. However, if the tool executes the same plan in the next cycle, for example when the service is currently at the final state, then the same state transitions will be occurred again during execution. This is clearly not a desired behaviour – indeed, the service will be stopped and then started again in every execution.

On the other hand, if the (original) solution plan is executed using our regression

execution, then the above state transitions will only be happened when the service is at the initial state of the plan. If the service is already at the final state, our algorithm will not make any change since the goal has been achieved.

5.2.6 Discovery Service

The discovery service plays a small but critical part in our choreography technique: it is only used to resolve the actual IP address and port number of particular agent's identifier. This address can be statically assigned by declaring the address and port explicitly in the specification, or could be dynamic, for example the IP address of an agent which controls a virtual machine (VM) will be assigned after the VM has been created, but not beforehand.

The previous simple cloud system is an example of the dynamic address resolution. Agent `vm1` might have a different IP address before and after migration between the two cloud infrastructures. If these clouds are managed by a single authority, then the address could be the same. However, if they are managed by two different authorities with different policies, then the address will be different[9]. Thus, the record in the discovery service should be updated after the migration has finished.

There are two approaches to implement the discovery service. First is centralised where the discovery service is running as an independent entity separately from the managed system. Although this could become a single point of failure, but we could have a set of discovery service instances in order to increase the robustness. The Domain Name System (DNS) service is a perfect example of this.

Second is fully distributed where every agent has an agents database to keep and resolve any identifier. The database is a simple key-value map where the agent's identifier is the key and the agent's IP address/port is the value. Every change on the agent's IP address are broadcasted to other agents. Whenever the agent receives such broadcast, then it will update the database by adding, removing, or altering the records. This technique is extremely lightweight and easily integrated into the agent runtime system. An alternative of this could be using a different technology such as Apache Zookeeper[Foundation, 2014b].

[9] In practice, two authorities (e.g. companies) may use different subnetworks – for the public network connected to the internet, two companies must not have the same subnetwork. Whenever a virtual machine is migrated between two companies, then a new IP address may need to be assigned so the machine can work properly within the new subnetwork.

5.2.7 Extending the Model

A limitation of our choreography technique is that it only constructs the BSig models from a single plan. This limits the number of possible current states which are covered by the existing models. In addition, since the model only has a single plan to a single goal, then the agents do not have any alternative plan to the goal, or alternative goal in case the execution is failed. Perhaps, a solution for this problem is plans merging where multiples plans to a single goal are merged into a single model. We leave this as part of the future works.

5.3 Summary

As summary, the first section describes how a planner can be integrated into an orchestration architecture to enable a close-loop control for unattended and self-healing system. It also gives the agent's architecture that clearly separates the configuration and the implementation. This makes the configuration becomes platform independent where multiple implementations in different platforms can work together seamlessly.

In the second section, a novel deployment technique called as *choreography* is introduced. The technique automatically constructs a set of reactive agents which choreograph the deployment of configuration changes without the need of central controller. By executing the persistent models using *cooperative regression reactive* (CRR) algorithm, the agents can form a self-healing system that can correct particular drifts from the desired state.

Chapter 6

Evaluation

This chapter presents evaluations of works that are described in the previous chapters. In the first section, the formal semantics of the SmartFrog (SF) language is evaluated by creating a compiler purely based on the semantics. The compiler was then used to compile some artificial specifications that contain edge cases, and specifications which are available in the distribution package of the SF production compiler. Its outputs were compared with the outputs of the SF production compiler.

The second section presents the evaluation of the performance of a domain independent compilation technique described in §4.1 for solving planning problems with extended goals. It measures the planning times and the planning coverage, and uses the MIPS-XXL [Edelkamp et al., 2006] planner as comparison.

In the third section, the technique described in §4.2 for generating the workflows of configuration tasks is evaluated using various configurations of typical systems in the cloud environment. The planning times and the planning coverage are presented for each use-case, and they are compared with MIPS-XXL planner.

The last section presents the Nuri configuration tool that: uses SFP language to define the configuration specification; has a planner to automatically generate the workflow; and implements the choreography to deploy the configuration changes. For evaluation, Nuri was used to deploy or reconfigure three real systems: Apache Hadoop, HP IDOLoop, and 3-tier web applications system, in the cloud environment.

6.1 Formal Semantics of SmartFrog Language

For evaluating SF semantics, we were able to translate fairly directly the formal semantics defined in §3.2 into a reference implementation using functional programming language Scala [sca, 2014], which is available as an open source software at:

`http://github.com/herry13/smartfrog-lang`

On the other hand, the semantics has also been independently implemented into two further compilers, purely from the semantics, and using different programming languages. One is implemented by the author using OCaml [oca, 2014], and another is implemented by Paul Anderson using Haskell [has, 2014]. Both are also available in the above repository. These are proving that the semantics is highly consistent and unambiguous, validating that it acts as a precise, independent reference for developers.

For further evaluation, we designed two experiments which aim to prove that the semantics with regards to the supported features, produces outputs that are equivalent to the production compiler. Note that we were using the production compiler version 3.18.016 which is available at the SmartFrog's website [HP Labs, 2014]. The first experiment was using artificial specifications which represent edge use cases, in particular for evaluating link reference resolution. The second experiment was using specifications of an example system which are available in the SmartFrog software distribution package.

For each experiment, the specifications were compiled using the production and our compilers. Afterwards, the outputs were then compared using **diff** program to find any difference. Note that the production compiler produces an output in the plain SmartFrog language representation, which is a specification without any prototype and link reference. Thus, we have implemented a helper function in our compilers that produces the same output to ease the comparison process.

6.1.1 Link Reference Resolution

This experiment is for evaluating the behaviour of link reference resolution. It used two specifications shown in figure 6.1 created by Patrick Goldsack. The specifications are compact, but they use almost all key features of the semantics such as the main component (`sfConfig`), prototypes and (forward) link references. Although they look different, but actually they are equivalent. This means that the compilation outputs

```
A extends { foo bar; }
sfConfig extends {
  test extends {
    bar 1;
    a1 extends A
  }
  bar 2;
  a2 test:a1;
}
```

(a)

```
A extends { foo bar; }
sfConfig extends {
  bar 2;
  a2 test:a1;
  test extends {
    bar 1;
    a1 extends A
  }
}
```

(b)

Figure 6.1: Two equivalent specifications with different orders of statements.

```
test extends  {
  bar 1;
  a1 extends  {
    foo 1;
  }
}
bar 2;
a2 extends  {
  foo 1;
}
```

(a)

```
test extends  {
  bar 1;
  a1 extends  {
    foo 1;
  }
}
bar 2;
a2 extends  {
  foo 1;
}
```

(b)

Figure 6.2: The outputs of the first specification (figure 6.1a) using (a) the production compiler and (b) our compiler.

```
bar 2;
a2 extends  {
  foo 1;
}
test extends  {
  bar 1;
  a1 extends  {
    foo 1;
  }
}
```

(a)

```
bar 2;
a2 extends  {
  foo 1;
}
test extends  {
  bar 1;
  a1 extends  {
    foo 1;
  }
}
```

(b)

Figure 6.3: The outputs of the second specification (figure 6.1b) using (a) the production compiler and (b) our compiler.

should be the same regardless of the choice of the compiler as well as the input specification.

The first specification (figure 6.1a) has a component (line 1) that has an attribute with a link reference value. It is used as the prototype of another component (line 5). Lines 2-9 define the main component which has three attributes (`test`, `bar` and `a2`), where the last one (line 8) has a value of link reference (`test:a1`) that refers another component's attribute defined in line 5.

The second specification (figure 6.1b) has the same statements but their orders are different – lines 7-8 of the first specification are moved to lines 3-4 in the second specification. This change of order should not affect the output since forward link references are supported by the semantics.

There are two critical values that test the resolving of link references. First, the link reference value of attribute `foo` of component `A` (line 1) should not be resolved since it is defined outside of the main component. Second, the final value of attribute `a1:foo` should be equal to `1`, not `2`. This is because the link reference value (`bar`), which is indirectly inherited from `A` through `test:a1`, should be resolved within the source component which `test:a1` and not within the target component. If the later is being used then the final value of `a1:foo` will be `2`.

The compilation outputs of the first specification are shown in figure 6.2, where the left one (figure 6.2a) was produced by the production compiler and the right one (figure 6.2b) was produced by our compiler. Other outputs of the compilation of the second specification are shown in figure 6.3, where the left (figure 6.3a) and the right (figure 6.3b) were produced by the production and our compilers respectively. You may notice that all outputs are identical including the orders of attribute names, which means that the semantics has an equivalent behaviour with the production compiler.

6.1.2 Specifications in the SmartFrog Distribution Package

This experiment is using a set of configuration specification files of example system which are available in the SmartFrog distribution package. However, since the semantics are only supporting the subset features of the production compiler, we then manually selected the files which are not using unsupported features such as functions and predicates. These selected, unmodified main specification files are shown in figure 6.4.

Each specification file describes the configuration of an example system. It contains not only the definition of the main component, but also file inclusion statements. Each of the included file contains a set of prototypes which are required by the main

No.	File	Lines	Size (Bytes)
1	org/smartfrog/examples/helloworld/example1.sf	49	1445
2	org/smartfrog/examples/helloworld/example1a.sf	37	1254
3	org/smartfrog/examples/helloworld/example1b.sf	37	1264
4	org/smartfrog/examples/helloworld/example1c.sf	36	1259
5	org/smartfrog/examples/helloworld/example1dist.sf	47	1410
6	org/smartfrog/examples/helloworld/example2.sf	43	1376
7	org/smartfrog/examples/helloworld/example3.sf	40	1271
8	org/smartfrog/examples/helloworld/example4.sf	64	1928
9	org/smartfrog/examples/helloworld/example5.sf	64	2102
10	org/smartfrog/examples/arithnet/example1.sf	52	1757
11	org/smartfrog/examples/arithnet/example2.sf	59	1930
12	org/smartfrog/examples/arithnet/example3.sf	99	3418

Figure 6.4: List of specification files from the SmartFrog distribution package which are used in the experiments.

No.	File	Lines	Size (Bytes)
1	org/smartfrog/components.sf	27	1041
2	org/smartfrog/examples/helloworld/printer.sf	30	1094
3	org/smartfrog/examples/helloworld/generator.sf	39	1408
4	org/smartfrog/functions.sf	293	7100
5	org/smartfrog/predicates.sf	127	3198
6	org/smartfrog/sfcore/prim/prim.sf	90	3947
7	org/smartfrog/sfcore/compound/compound.sf	36	1272
8	org/smartfrog/sfcore/workflow/combinators/detachingcompound.sf	40	1681
9	org/smartfrog/examples/arithnet/netComponents.sf	103	2756

Figure 6.5: List of included specification files from the SmartFrog distributed package.

specification. Both the production and our compilers parses the main and included files in order to produce the final description. These included files are shown in figure 6.5

Due to copyright issue, we do not include the source code of the specification files in this thesis. The reader is advised to download the files directly from [HP Labs, 2014]:

`http://sourceforge.net/p/smartfrog/svn/HEAD/tree/trunk/core/smartfrog/src/`

Every main specification file was compiled with the production and our compiler, and the outputs were then directly compared with **diff** program to detect differences. For example, the following is the output generated our compiler when file #8 (in figure 6.4) is given as the input:

```
sfCodeBase "default";
sfClass "org.smartfrog.sfcore.workflow.combinators.DetachingCompoundImpl";
detachDownwards true;
detachUpwards true;
autoDestruct true;
pair1 extends  {
  sfCodeBase "default";
  sfClass "org.smartfrog.sfcore.compound.CompoundImpl";
  messages ["hello", "world", "again"];
  frequency 10;
  g extends  {
    sfCodeBase "default";
    sfClass "org.smartfrog.examples.helloworld.GeneratorImpl";
    frequency 10;
    messages ["hello", "world", "again"];
    printer LAZY p;
  }
  p extends  {
    sfCodeBase "default";
    sfClass "org.smartfrog.examples.helloworld.PrinterImpl";
  }
}
pair2 extends  {
  sfCodeBase "default";
  sfClass "org.smartfrog.sfcore.compound.CompoundImpl";
  messages ["this␣is␣a", "boring", "set␣of␣strings"];
  frequency 5;
  g extends  {
    sfCodeBase "default";
    sfClass "org.smartfrog.examples.helloworld.GeneratorImpl";
    frequency 5;
    messages ["this␣is␣a", "boring", "set␣of␣strings"];
    printer LAZY p;
  }
  p extends  {
    sfCodeBase "default";
    sfClass "org.smartfrog.examples.helloworld.PrinterImpl";
  }
}
```

Using the same file, the production compiler generated the following output:

```
sfCodeBase "default";
sfClass "org.smartfrog.sfcore.workflow.combinators.DetachingCompoundImpl";
detachDownwards true;
detachUpwards true;
autoDestruct true;
pair1 extends  {
  sfCodeBase "default";
  sfClass "org.smartfrog.sfcore.compound.CompoundImpl";
  messages [|"hello", "world", "again"|];
  frequency 10;
  g extends  {
    sfCodeBase "default";
    sfClass "org.smartfrog.examples.helloworld.GeneratorImpl";
```

```
14      frequency 10;
15      messages [|"hello", "world", "again"|];
16      printer LAZY p;
17    }
18    p extends  {
19      sfCodeBase "default";
20      sfClass "org.smartfrog.examples.helloworld.PrinterImpl";
21    }
22  }
23  pair2 extends  {
24    sfCodeBase "default";
25    sfClass "org.smartfrog.sfcore.compound.CompoundImpl";
26    messages [|"this␣is␣a", "boring", "set␣of␣strings"|];
27    frequency 5;
28    g extends  {
29      sfCodeBase "default";
30      sfClass "org.smartfrog.examples.helloworld.GeneratorImpl";
31      frequency 5;
32      messages [|"this␣is␣a", "boring", "set␣of␣strings"|];
33      printer LAZY p;
34    }
35    p extends  {
36      sfCodeBase "default";
37      sfClass "org.smartfrog.examples.helloworld.PrinterImpl";
38    }
39  }
```

Notice that the above outputs are identical.

We did the same process for other files and automatically compare the outputs using **diff**. Based on the results, **diff** cannot find any difference between the output produced by our compiler with the output produced by the production compiler. This proves that the semantics with regards to the supported features, produces outputs that are equivalent to the production compiler. All outputs of both compilers can be found in:

`https://github.com/herry13/smartfrog-lang/tree/thesis/test/sf-dist/`

6.2 Planning with Extended Goal

These experiments aim to measure the performance of our compilation technique for solving planning problems with extended goals[1]. The performance is measured using two metrics: the required time to solve the problem (planning time), and the number of problems that can be solved within given deadline (planning coverage).

6.2.1 Design of Experiments

The experiments used three problem domains from the 5th International Planning Competition (IPC-5) which are Openstacks, Rovers and Storage. Each domain consists of 20 problems defined in PDDL, each of which contain not only the hard extended goals, but also the soft ones (preferences).

The experiments used three problem domains from the 5th International Planning Competition (IPC-5) which are Openstacks, Rovers and Storage. Each domain consists of 20 problems defined in PDDL, each of which contain not only the hard extended goals, but also the soft ones (preferences). A hard goal is a strict goal that must be achieved by a plan. On the other hand, a plan may not achieve a soft goal, but this will decrease the quality of the plan.

Since our work is only focusing on the hard extended goals, then a script was used for randomly selecting and converting the soft extended goals to hard ones. The script performed random selection 100 times for each problem. Because there are 20 problems for each domain, then after random selection, the script generated 2000 problems which are grouped into 20 datasets based on their original problem – each dataset has 100 problems. It is possible that there is no solution plan for the generated problem since the randomly selected soft goals may conflict with each other.

We implemented the compilation technique described in §4.1 into FastDownward[2] [Helmert, 2006] planner by modifying its PDDL-to-FDR translator component[3], which is then called as FDT. We selected h^{FF} [Hoffmann and Nebel, 2001] as the heuristic technique of FDT. For comparison, we chosed MIPS-XXL [Edelkamp and Jabbar, 2008] planner which is a participant of IPC-5. Note that MIPS-XXL is using h^{FF} for calculating the heuristics.

[1] The evaluation using problems from system configuration domain will be presented in §6.3.

[2] The original FastDownward planner does not support extended goals.

[3] The FastDownward planner consists of three components: the PDDL-to-FDR translator, the pre-processor, and the search engine.

All planners were using the same configurations. Every experiment was running on a server with Intel Xeon CPU 16 cores, 2.6 GHz for each core, 48 GB of memory, and Linux operating system. However, every planner was set to only use one CPU core and maximum 24 GB of memory. For every problem, the planner was given maximum of 30 minutes to solve it. Every run will result in *solved*[4] or *unsolved*[5]. Every plan generated either by FDT or MIPS-XXL was checked using VAL [Howey et al., 2004], which is the standard plan validator software for PDDL planning problem. If the plan is invalid, then the planner is assumed to have failed in finding the solution plan.

6.2.2 Problem Domains

The following paragraphs describe the problem domains.

Openstacks The Openstacks domain is based on the *minimum maximum* open stacks combinatorial optimisation problem. The domain states a scenario where a manufacturer has to make products based on a number of orders, each of which is for a combination of different types of products. The production activities are constrained by a condition where the production process is sequential – only one product can be finished at a single production batch. An order is said to be "open" if all products have not been delivered, and it will require a "stack" at particular step to store the finished products and wait until all types of product have been produced by the next production batch. The problem is to arrange the making of different products such that the maximum number of stacks that are being used simultaneously is minimised. Examples of extended goals that can be found in this domain are: the number of orders/products that should be delivered; a set of stacks that should not be used during production process.

Rovers The Rovers domain models the planning of activities of one or more autonomous rovers for exploring a planet to obtain material samples such as rocks or soils from particular waypoints, or to capture images of some objects. The problems contain various hard and soft extended goals, for examples: the samples must be obtained within a particular order; every rover has a limited sample storage capacity and can only visit a limited number of areas.

[4]The plan is found, or the planner stated that there is no solution plan.
[5]It is timeout (the planner exceeds 30 minutes deadline), out of memory, or invalid plan.

Storage The Storage domain models a typical transportation problem involving activities to move items from one to another place using particular equipment. The items are crates in containers, and one or more hoists can be used to move them to storage spaces (depots). The problems are similar with puzzle games (e.g. Sokoban) where the depot is divided into several areas. The movements of the hoist is restricted in the depot where it can only move between adjacent areas, and can only enter and leave the depot from/to particular area. The containers are located outside the depot where the hoist can move unrestricted. The examples of the extended goals are: some crates must be stored in particular area and/or within particular order; some crates can only be lifted using particular hoist.

6.2.3 Results and Analysis

6.2.3.1 Openstacks

The total number of solved Openstacks problems based on the planning time of the planners are illustrated in figure 6.6. The dark-grey line is representing FDT, while the light-grey line is representing MIPS-XXL. After 30 minutes, FDT solved 1776 problems, which are 285 more than MIPS-XXL that solved 1491 problems.

Figure 6.7 shows the number of solved problems for each dataset. It shows that FDT outperformed MIPS-XXL on every dataset. Interesting results are shown in dataset p19 and p20 where MIPS-XXL did not solve any problem - FDT solved 81 and 74 problems for p19 and p20 respectively. The log files showed that MIPS-XXL exited unexpectedly due to some internal errors when trying to solve the problems of these two datasets. Unfortunately, we cannot find a solution to address the errors. Thus, from 509 unsolved problems, 200 of them were caused by these errors while 309 were caused by time out. On the other side, FDT cannot solve 224 problems because of time out.

Figure 6.8 shows the minimum, the maximum and the average planning time of FDT and MIPS-XXL based on the problems solved by both planners. The problems which were not solved by both planners are excluded – there is no data for dataset p19 and p20 because MIPS-XXL did not solve any problem in these datasets. FDT has a better average planning time in every dataset where the differences are between about 0.5-13 seconds. FDT also has a better maximum planning almost on every dataset except p13 and p15.

Clearly, the figures show that FDT has a better overall performance either in plan-

ning time or coverage comparing to MIPS-XXL. This fact is shown in figure 6.6 where about 1400 problems were solved by FDT within the first 10 seconds, while MIPS-XXL required 60 seconds to reach that numbers. On the other hand, FDT has a better planning coverage where it solved 19 percent more problems than MIPS-XXL within 30 minutes.

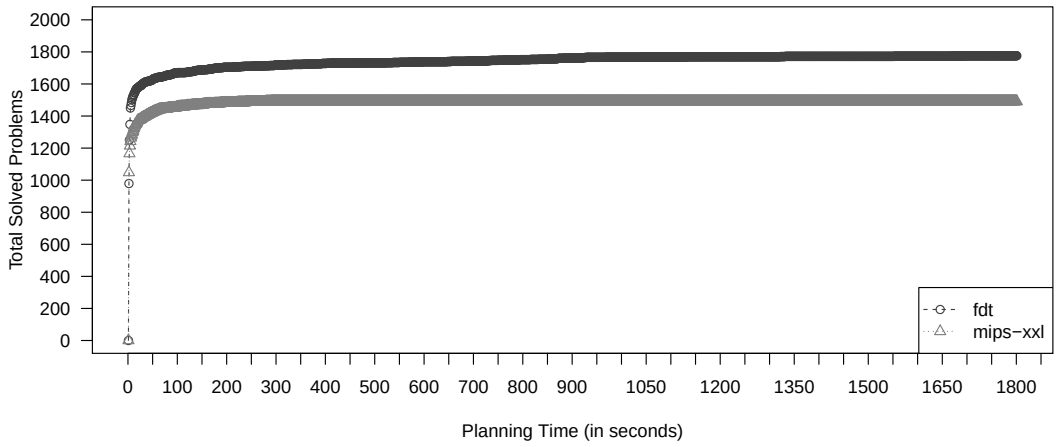


Figure 6.6: The accumulated number of solved problems for Openstacks domain based on planning time FDT (dark-grey) and MIPS-XXL (light-grey). The total number of problems is 2000 (20 datasets, each of which has 100 problems). A higher line means that the planner can solve more problems in shorter time than another.

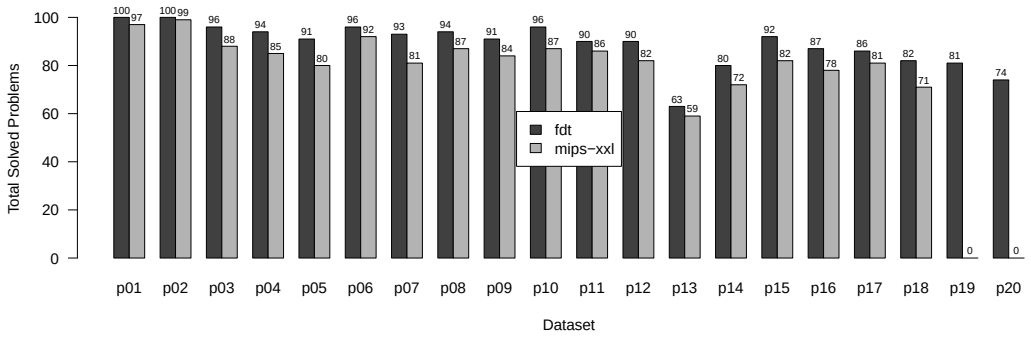


Figure 6.7: The number of solved problem per dataset for Openstacks domain using FDT (dark-grey) and MIPS-XXL (light-grey). There are 20 datasets i.e. p01-p20, each of which has 100 problems. Each dataset differs from the others on the number of products/orders that should be delivered and the number of stacks that can be used during production.

Dataset	Solved by Both (Total Problems)	FDT			MIPS-XXL		
		Min(s)	Avg(s)	Max(s)	Min(s)	Avg(s)	Max(s)
p01	97 (100)	0.17	**0.25**	1.95	0.05	0.75	48.93
p02	99 (100)	0.17	**0.29**	1.91	0.05	1.35	53.6
p03	88 (100)	0.31	**2.15**	91.26	0.14	4.73	117.3
p04	85 (100)	0.33	**1.53**	22.04	0.12	6.67	118.69
p05	80 (100)	0.44	**1.39**	26.12	0.23	3.77	101.67
p06	92 (100)	0.42	**2.13**	45.31	0.24	6.77	187.35
p07	81 (100)	0.46	**1.99**	33.1	0.2	3.94	80.22
p08	87 (100)	0.4	**3.14**	49.66	0.21	10.74	159.06
p09	84 (100)	0.43	**4.65**	131.04	0.2	8.84	242.89
p10	87 (100)	0.41	**2.12**	52.67	0.2	9.28	279.79
p11	86 (100)	0.46	**2.07**	48.42	0.19	7.48	276.46
p12	82 (100)	0.5	**2.43**	49.65	0.23	6.62	183.4
p13	55 (100)	0.69	**27.04**	792.6	0.44	29.66	236.78
p14	72 (100)	0.6	**4.1**	75.74	0.35	12.08	289.57
p15	82 (100)	0.89	**5.87**	210.98	0.53	8.66	197.25
p16	78 (100)	0.78	**5.72**	179.71	0.52	18.03	250.78
p17	81 (100)	0.76	**2.15**	21.68	0.52	8.42	154.2
p18	70 (100)	0.92	**3.17**	50.58	0.52	6.41	116.92
p19	0 (100)	-	-	-	-	-	-
p20	0 (100)	-	-	-	-	-	-
	1486	0.17	**3.48**	792.6	0.05	7.98	289.57

Figure 6.8: The minimum, average and maximum planning time for Openstacks problems which are solved by both FDT and MIPS-XXL. Note that the problems which were not solved by both planners are excluded. Each dataset differs from the others on the number of products/orders that should be delivered and the number of stacks that can be used during production.

6.2.3.2 Rovers

The total number of solved Rovers problems, based on the planning time, are illustrated in figure 6.9. The dark-grey line is representing FDT, while the light-grey line is representing MIPS-XXL. After 30 minutes, FDT solved 524 problems, which are 308 more problems than MIPS-XXL that solved 216 problems.

Figure 6.7 shows the number of solved problems for each dataset. It shows that FDT outperforms MIPS-XXL on 12 datasets which are p01-p08, p10, p12-p13 and p16. While on the other 8 datasets (p09, p11, p14-p15, p17-p20), no planner can solve any problem because of time out or out of memory.

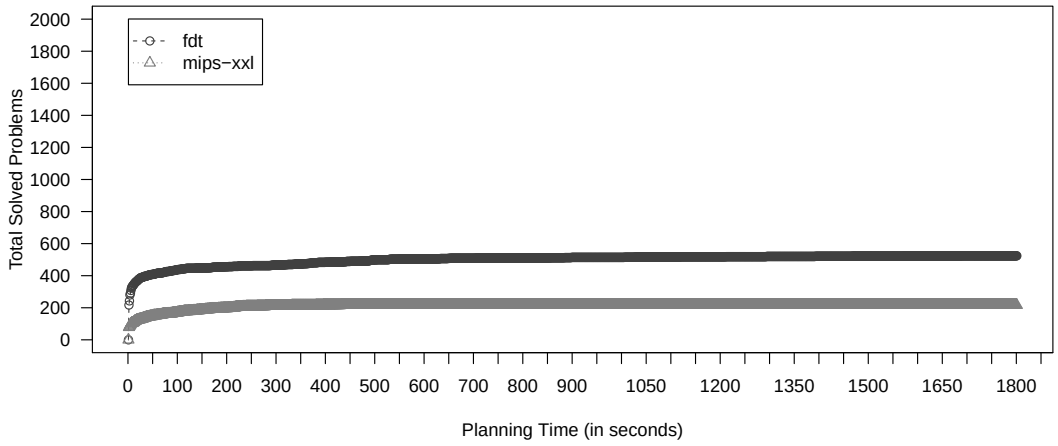


Figure 6.9: The accumulated number of solved problem for Rovers domain based on planning time FDT (dark-grey) and MIPS-XXL (light-grey). The total number of problems is 2000 (20 datasets, each of which has 100 problems). A higher line means that the planner can solve more problems in shorter time than another.

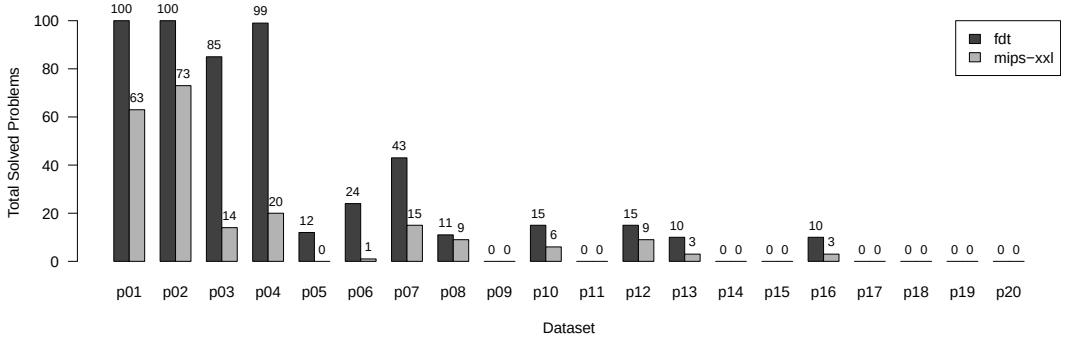


Figure 6.10: The number of solved problem per dataset for Rovers domain using FDT (dark-grey) and MIPS-XXL (light-grey). There are 20 datasets i.e. p01-p20, each of which has 100 problems. Each dataset differs from others on the numbers of rovers, objects, and waypoints.

Figure 6.11 shows the minimum, the maximum and the average planning time of FDT and MIPS-XXL based on the problems solved by both planners[6]. There are no results for datasets p05, p09, p11, p14, p15 and p17-p20 because MIPS-XXL and FDT did not solve any problem in these datasets. Overall, FDT has a significant average planning time which is less by about 50 seconds.

[6]The problems which were not solved by both planners are excluded

Dataset	Solved by Both	FDT			MIPS-XXL		
	(Total Problems)	Min(s)	Avg(s)	Max(s)	Min(s)	Avg(s)	Max(s)
p01	63 (100)	0.08	**4.96**	56.25	0.01	73.16	278.51
p02	73 (100)	0.08	**0.69**	5.0	0.01	50.47	286.85
p03	14 (100)	0.11	**0.36**	0.5	0.03	173.32	430.8
p04	20 (100)	0.08	**0.35**	2.15	0.02	37.31	212.64
p05	0 (100)	-	-	-	-	-	-
p06	1 (100)	5.38	5.38	5.38	0.06	**0.06**	0.06
p07	15 (100)	0.12	**0.18**	0.29	0.03	18.61	277.82
p08	9 (100)	0.19	**0.3**	0.47	0.07	2.16	17.11
p09	0 (100)	-	-	-	-	-	-
p10	6 (100)	0.43	**0.81**	2.14	0.2	1.83	3.02
p11	0 (100)	-	-	-	-	-	-
p12	8 (100)	0.21	0.49	1.45	0.08	**0.32**	1.41
p13	3 (100)	0.63	122.26	365.25	4.6	**8.43**	15.2
p14	0 (100)	-	-	-	-	-	-
p15	0 (100)	-	-	-	-	-	-
p16	3 (100)	1.17	**1.39**	1.69	0.89	4.86	11.25
p17	0 (100)	-	-	-	-	-	-
p18	0 (100)	-	-	-	-	-	-
p19	0 (100)	-	-	-	-	-	-
p20	0 (100)	-	-	-	-	-	-
	215	0.08	**3.56**	365.25	0.01	54.97	430.8

Figure 6.11: The average planning time for Rovers problems which are solved by both FDT and MIPS-XXL. Note that the problems which were not solved by both planners are excluded. Each dataset differs from others on the numbers of rovers, objects, and waypoints.

Overall, FDT outperforms MIPS-XXL either in planning time or coverage. This fact is clearly shown in figure 6.9 where FDT solved about 400 problems within only 20 seconds, while MIPS-XXL never reached that number after 30 minutes. On the other hand, FDT solved more than twice the number of problems solved by MIPS-XXL. In addition, MIPS-XXL was never solving more problems than FDT on every dataset.

6.2.3.3 Storage

The overall number of solved problems of the planners are summarised in figure 6.12. The dark-grey and light-grey are representing FDT and MIPS-XXL respectively. In 30

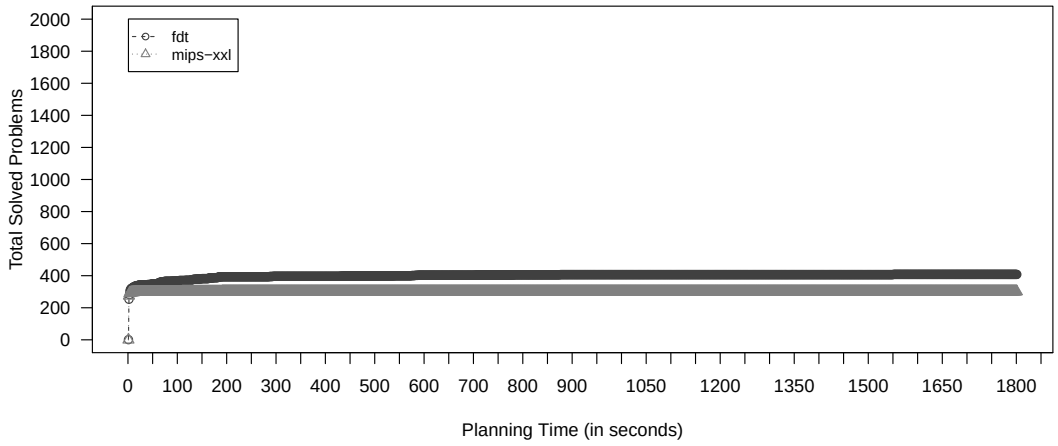


Figure 6.12: The number of solved problem for Storage domain based on planning time FDT (dark-grey) and MIPS-XXL (light-grey). The total number of problems is 2000 (20 datasets, each of which has 100 problems). A higher line means that the planner can solve more problems in shorter time than another.

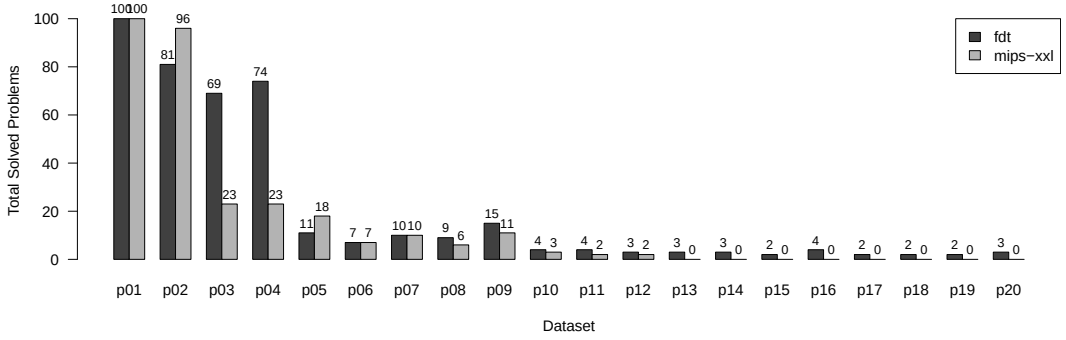


Figure 6.13: The number of solved problems per dataset for Storage domain using FDT (dark-grey) and MIPS-XXL (light-grey). There are 20 datasets i.e. p01-p20, each of which has 100 problems. Each dataset differs from the others on the number of crates, hoists, and storages.

minutes, FDT solved 408 problems, which are 107 more than MIPS-XXL that solved 301 problems.

The details of the number of solved problems for each dataset are shown in figure 6.13. It shows that FDT can solve some problems on every dataset. On the other hand, MIPS-XXL can only solve problems in 12 datasets which are p01-p12, and it cannot solve any problem in datasets p13-p20 because of time out.

Dataset	Solved by Both (Total Problems)	FDT			MIPS-XXL		
		Min(s)	Avg(s)	Max(s)	Min(s)	Avg(s)	Max(s)
p01	100 (100)	0.11	0.13	0.21	0.02	**0.03**	0.04
p02	78 (100)	0.15	0.2	0.28	0.03	**0.09**	0.23
p03	11 (100)	0.22	0.25	0.29	0.06	**0.07**	0.08
p04	15 (100)	0.32	0.46	0.93	0.09	**0.13**	0.24
p05	10 (100)	0.75	**0.89**	1.55	0.24	19.19	189.52
p06	5 (100)	0.96	1.0	1.04	0.31	**0.35**	0.39
p07	6 (100)	1.27	1.68	3.1	0.47	**0.53**	0.65
p08	5 (100)	1.62	9.7	26.37	0.59	**0.73**	0.94
p09	9 (100)	3.15	4.79	8.98	1.08	**1.36**	1.7
p10	3 (100)	3.84	31.69	86.97	1.45	**1.57**	1.73
p11	2 (100)	4.79	7.44	10.09	1.89	**1.9**	1.91
p12	2 (100)	5.64	5.94	6.23	2.11	**2.19**	2.27
p13	0 (100)	-	-	-	-	-	-
p14	0 (100)	-	-	-	-	-	-
p15	0 (100)	-	-	-	-	-	-
p16	0 (100)	-	-	-	-	-	-
p17	0 (100)	-	-	-	-	-	-
p18	0 (100)	-	-	-	-	-	-
p19	0 (100)	-	-	-	-	-	-
p20	0 (100)	-	-	-	-	-	-
	246	0.11	1.12	86.97	0.02	**0.97**	189.52

Figure 6.14: The average planning time for Storage problems which are solved by both FDT and MIPS-XXL. Note that the problems which were not solved by both planners are excluded. Each dataset differs from the others on the number of crates, hoists, and storages.

There is an interesting result from this domain. Unlike in the Openstacks and the Rovers domains, MIPS-XXL can solve more problems than FDT in some datasets which are p02 and p05. In dataset p02, FDT generated 22 invalid plans and never timed out. While in dataset p05, FDT was time out on some problems.

Figure 6.14 shows the minimum, the maximum and the average planning time of FDT and MIPS-XXL based on the problems solved by both planners[7]. There is no data for dataset p13-p20 because MIPS-XXL cannot solve any problem in those datasets. Overall, there is much difference on the average planning time between both planners.

Overall, FDT slightly outperforms MIPS-XXL for the Storage domain. This fact

[7]The problems which were not solved by both planners are excluded

is shown in figure 6.9 where FDT solved 107 more problems than MIPS-XXL.

6.2.4 Discussion

The results in §6.2.3.1, §6.2.3.2 and §6.2.3.3 indicate that FDT has better planning time and coverage in all domains comparing to MIPS-XXL. This shows that the compilation technique used in FDT (see §4.1) is more efficient comparing to MIPS-XXL. Note that both planners are basically using the same heuristics technique i.e. h^{FF} [Hoffmann and Nebel, 2001].

[Edelkamp et al., 2006] describes that MIPS-XXL compiles every state-trajectory constraint into a Büchi automaton and generates the corresponding PDDL codes to simulate the automaton's state-transition. [Edelkamp, 2006] provides more details about this compilation scheme with a notable statement: "the size of the Büchi automaton for a given state-trajectory constraint formula can be exponential in the length of the formula". Unfortunately, this critical property can significantly degrade the performance of MIPS-XXL for a problem with a large number of state-trajectory constraints. This is because MIPS-XXL will generate new actions in order to perform the automaton's state-transition. Since the size of the automaton can be exponential, then the number of new actions can be exponential as well.

On the other hand, the compilation scheme implemented in FDT (see §4.1) is using a different approach: every state-trajectory constraint is compiled into a finite automaton and only one new action is generated to performed the automaton's state transitions using conditional effects. Thus, the number of new actions after compilation is linear to the number of constraints. We suspect that this is the reason why FDT performed better than MIPS-XXL.

Domain	Average Planning Time[8]		Solved Problems	
	FDT	MIPS-XXL	FDT	MIPS-XXL
Openstacks	**3.48**	7.98	**1776**	1491
Rovers	**3.56**	54.97	**524**	216
Storage	1.12	**0.97**	**408**	301

Figure 6.15: The summary of the average planning time and the number of solved problems by FDT and MIPS-XXL for all domains.

6.2.5 Summary

Figure 6.15 summarises the results of experiments. It shows that the compilation technique described in §4.1 is a promising approach to solve a planning problem with extended goals. By implementing the technique into the FastDownward planner (FDT), it outperforms MIPS-XXL in three IPC domains which are Openstacks, Rovers and Storage.

[8]The average planning time for problems solved by FDT and MIPS-XXL. The problems which are not solved by both planners are excluded.

6.3 Planning Configuration Changes

This section describes experiments that aim to measure the performance of the Nuri planner that implements the technique described in §4.2 to solve system configuration tasks. We chose MIPS-XXL planner for comparison. Two metrics were used to compare the planners' performance i.e. the planning time, and the planning coverage (the number of solved problems within given deadline). For the configuration tasks, we used three artificial systems that should be configured in two different scenarios.

6.3.1 Design of Experiments

6.3.1.1 Scenarios

The experiments were designed to focus on solving the system configuration tasks in the cloud environment. We used two different scenarios for each use-case i.e. the *cloud-deployment* and the *cloud-burst* scenarios.

In the cloud-deployment scenario, we deploy a system from scratch to a cloud infrastructure. Figure 6.16 illustrates the current (left) and the desired (right) configuration state of an example system consisting of two virtual machines and two services. Since there are dependencies between the components (the virtual machine with the service that runs on it, and between services), then the deployment process of the system must preserve particular global constraints, for example: service `app1` must be running before service `app0` is running.

On the other side, figure 6.17 illustrates the current (left) and the desired (right)

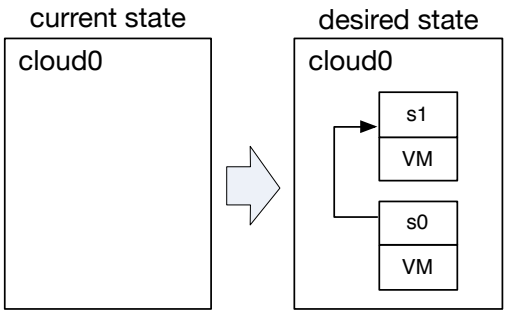


Figure 6.16: The cloud deployment scenario where the left and the right are the current and the desired configuration state of the system respectively. The arrows are representing the dependencies between services. `s0` $\rightarrow$ `s1` means that `s0` depends on `s1`. Thus, `s1` must be started before `s0`, and `s1` must be stopped after `s0`.

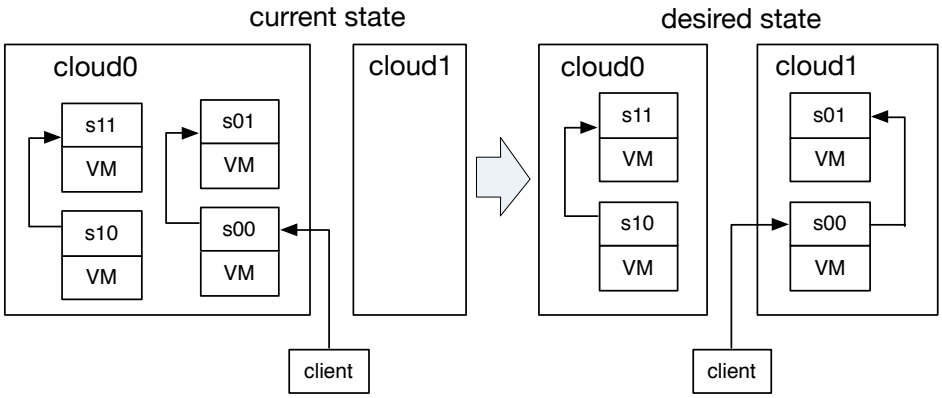


Figure 6.17: The cloud burst scenario where the left and the right are the current and the desired configuration state of the system respectively. The arrows are representing the dependencies between services (`s00` $\rightarrow$ `s01` means that `s00` depends on `s01`, thus, `s01` must be started before `s00`, or `s01` must be stopped after `s00`), or between the client and the service (the client must always refer to a running service).

configuration state of an example system in the cloud-burst scenario. This scenario assumes that a company has a private cloud infrastructure (`cloud0`) which runs various services to serve its 24-hours operations. One of them is the most important one because it processes all financial transactions from the company's branch offices. Thus, the system administrator has prepared a backup system installed in the private cloud as well. Unfortunately, due to the limited resource of the physical machines, the company's own private cloud infrastructure is not capable of serving the spikes in demand which usually happens on the last three days of each month. Therefore, before the spike period, the administrator plans to migrate the main system temporarily to the public cloud (`cloud1`) to minimise its response time. Each configuration task of this scenario will have two cloud infrastructures and two identical systems, where one of the system will be migrated to another cloud.

Although both scenarios look similar, but they have different characteristic. The cloud-burst tasks involve a reconfiguration of an existing system that, even for a small system, can yield a "butterfly" effect of configuration changes due to the dependencies between the resources. On the other hand, this effect usually does not exist when deploying the system from scratch e.g. cloud-deployment tasks.

The planners were asked to solve configuration tasks that configure three artificial systems i.e. system-A, system-B, and system-C in both cloud deployment and cloud burst scenarios. Every configuration task is defined in SFP where the specification

consists of the resource models (schemata), the current and the desired state of the system, and the global constraints – the constraints that should be maintained during configuration changes.

6.3.1.2 Nuri Planner

We have developed the Nuri planner which is capable of finding a solution plan for SFP configuration tasks. The planner consists of three components: the compiler, the search engine, and the postprocessor. The compiler implements the SFP semantics described in §3.4 as well as the translation technique describe in §4.2 that translates a configuration task into a classical planning problem. In addition, the compiler also implements the compilation scheme describe in §4.1. The search engine is taken from the FastDownward [Helmert, 2006] classical planner (without any modification). And the postprocessor implements the postprocessing technique described in §4.2. All components are implemented in Ruby, except the search engine which is implemented in C++. Note that the Nuri planner has implemented the parallel multi-heuristics search technique described in §4.2.3. However, in this experiment, since we will compare it with another planner that only supports a single heuristic, then for fairness we disabled this parallel capability so that only one heuristic can be used at a time.

Since Nuri's search engine supports several heuristics, we created two Nuri planner instances that use two different heuristics: 1) Nuri^{FF} that uses h^{FF} heuristic [Hoffmann and Nebel, 2001], and 2) Nuri^{LM} that uses h^{LM} heuristic [Richter et al., 2008]. There are several reasons why we chose these heuristics:

- Both heuristics have been empirically proved to have good performances on various planning domains. This is supported by the planners which won the International Planning Competition (IPC): FF [Hoffmann and Nebel, 2001] (h^{FF}) in 2000, FASTDOWNWARD [Helmert, 2006] (h^{FF}) in 2004, and LAMA (h^{LM} and h^{FF}) [Richter and Westphal, 2010] in 2008 and 2011.

- From our observations, we found many mutual exclusive actions in configuration tasks. For examples: all virtual machines can be created or deleted simultaneously; software packages can be installed simultaneously on different machines. h^{FF} is pretty good on estimating the heuristic value (i.e. an estimated distance of an action to the goal) of these mutual exclusive actions. This is supported by the fact that h^{FF} computes the heuristics by solving a "relaxed" planning problem

using the planning-graph technique [Blum and Furst, 1997a][9], which allows the planner to select two or more mutual exclusive actions simultaneously in a single step. Thus, it is highly likely that these mutual exclusive actions will have the same heuristic values. This is a good estimation since it is similar to the characteristic of the solution plan of the original problem. In addition, h^{FF} will set the heuristic value to infinite if the action is not part of the solution of the relaxed problem, which helps the planner to avoid visiting superfluous states.

- Another thing that we found from the observations is that the configuration tasks have a common characteristic: a particular fact must be obtained before or after another. For examples: a virtual machine must be created before it is running; a software package is installed after its virtual machine is running and before the service is running. This characteristic is similar to the definition of *landmarks* (see §2.2.4.2). Since h^{LM} computes the heuristics based on landmarks, then we can expect that it will produce good heuristic values.

6.3.1.3 Settings

We chose MIPS-XXL planner for comparison[10]. However, since it only accepts problems defined in PDDL3, then we also specified every configuration task in PDDL3. For fairness, we tried to define every task as similar as possible in PDDL3 and SFP. For example, in both representations, we define the same number of objects and use similar logic formulas in the actions' preconditions and the global constraints. The examples of configuration tasks in PDDL3 and SFP can be found in appendix C. Note that MIPS-XXL only supports one heuristic i.e. h^{FF} .

Besides MIPS-XXL, there is another planner that can solve a planning problem with global constraints i.e. SGPlan [Hsu et al., 2006]. However, it cannot solve any problem except the IPC-5 domains. We does not know the reason because it is only distributed in binary. We are not aware of other planners that natively supports PDDL3.

In the experiments, every dependency is defined as an implication formula of the global constraints. For example, if service *a* depends on *b* (*b* must be started before *a*, or *b* must be stopped after *a*), then we define the following implication formula in the SFP configuration task:

[9] Although the planning-graph technique can solve a planning problem in exponential-time, but it is guaranteed that it can solve a "relaxed" planning problem in polynomial [Hoffmann and Nebel, 2001].

[10] Since MIPS-XXL can only generate a sequential plan, then for the sake of fairness, we also disabled the partial-order plan generator of the Nuri planner during experiments. The partial-order plan generator can be easily activated by passing a particular option to the Nuri planner.

```
1 global {
2   ...
3   if a.running = true then b.running = true
4   ...
5 }
```

On the other hand, the dependency is defined also as implication formula in the PDDL task:

```
1 (:constraints (and
2     ...
3     (always (imply (running a) (running b)))
4     ...
5 ))
```

All experiments are using the same settings. Every experiment was running on a server with Intel Xeon CPU 8 cores, 2.4 GHz, 48 GB of memory, and Linux operating system. However, Nuri and MIPS-XXL were set to use one CPU core and a maximum of 24 GB of memory. For every task, every planner was given a maximum of 8 hours to solve it.

6.3.2 Description of the Systems

6.3.2.1 System-A

System-A is an artificial system inspired by a typical system consisting of m identical subsystems so that the requests can be processed in parallel. Each subsystem has multilayer application services. In the front end, there is a load balancer service which equally distributes the requests among the back end subsystems.

Figure 6.18 illustrates the architecture of system-A, which has a set of application services which are are grouped "vertically" into one or more subsystems, where there are dependencies (arrows) between services in the same group. The first service of every subsystem is connected to the load balancer service. Every service is running on a virtual machine (VM). Thus, there is an internal dependency between the service with its VM: the service can only be installed or started after the VM has been created and started, and the VM can only be stopped if the service has been stopped.

We created different configurations of system-A, each of which has different combination of the number of subsystems (m) and the number of layers (n). The example of configuration tasks for the cloud-deployment scenario of system-A with 2 subsystems (each has 2 services) in SFP and PDDL are available in §C.1.1.1 and §C.1.1.2

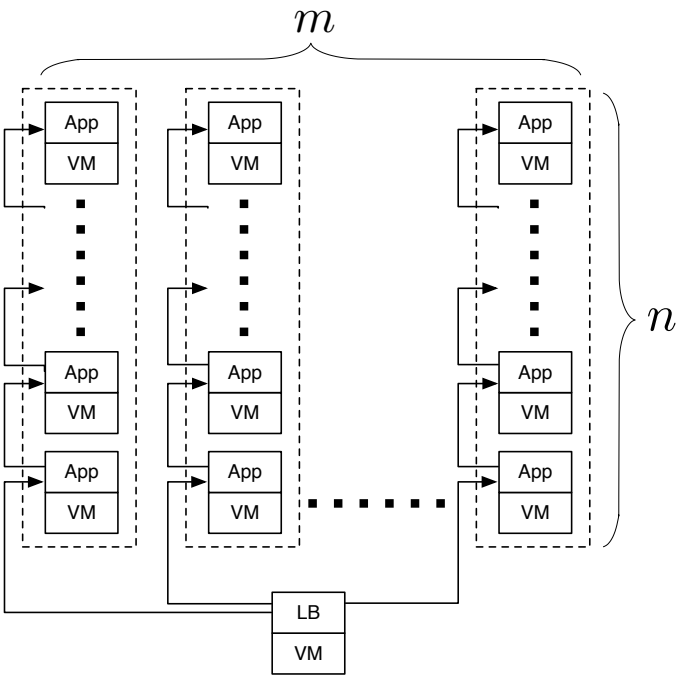


Figure 6.18: **System-A**: a cloud-based system where there are m subsystems, each of which has n application layers. All subsystems are connected to a load balancer (LB) as the service interface to the user.

respectively. While the example of tasks for the cloud-burst scenario are available in §C.1.2.1 and §C.1.2.2 respectively.

6.3.2.2 System-B

System-B is an artificial system inspired by a typical multilayer system where services at particular layer depend on services at the next layer. Figure 6.19 illustrates the architecture of system-B. Unlike system-A, the services are divided horizontally into several layers. Each layer is a subsystem consisting of one load balancer and a set of application services. These services are not connected to each other, but they are connected to the current layer's load balancer and the next layer's load balancer (for accessing services at the next layer). These connections represent the dependencies (arrows) between the services, for example: the application services must be started before their load balancer is started. Every service is running on top of a VM. Thus, the VM must be created before the service can be installed and started.

We created different configurations of system-B, each of which has different combination of the number of layers (n) and the number of application services per layer

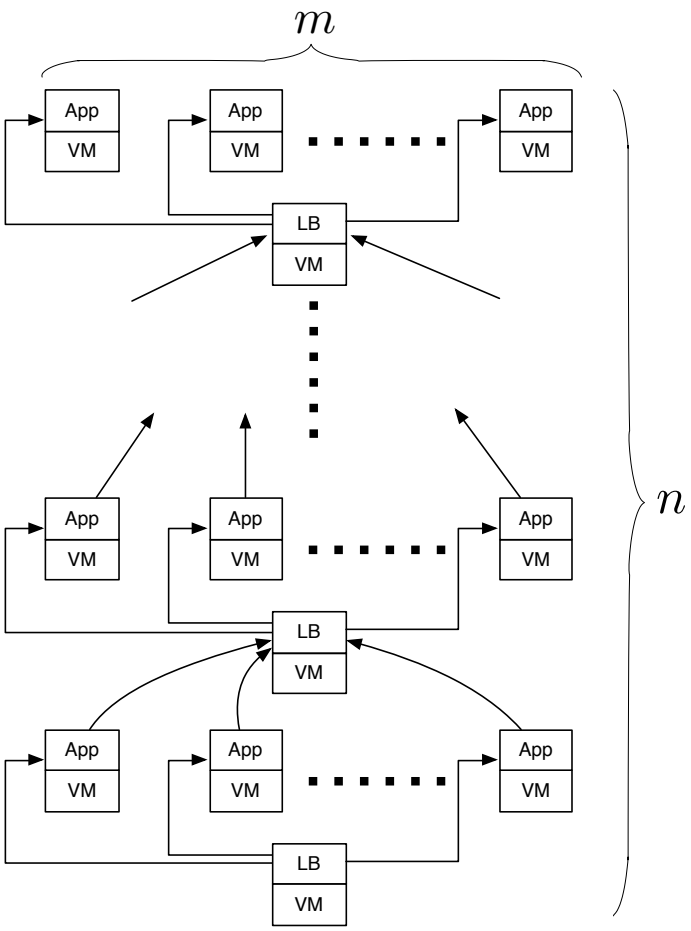


Figure 6.19: **System-B**: a cloud-based system where there are n layers of subsystem, each of which consists of one load balancer (LB) and m application services.

(m). §C.2.1.1 and §C.2.1.2 show the example of configuration tasks for the cloud-deployment scenario of system-B with 2 layers (each has 2 services) in SFP and PDDL respectively. While §C.2.2.1 and §C.2.2.2 show the example of configuration tasks for the cloud-burst scenario in SFP and PDDL respectively.

6.3.2.3 System-C

System-C is the last artificial system that has n application services whose dependencies were randomly generated. For the system with n services, a script generated a random directed acyclic graph with $(n-1)$ nodes. Then the script added an extra node to the graph as the front-end service of the system (the graph has in total n nodes), where the extra node has out-edges to other nodes whose in-degree is equal to zero. The graph edges were then used to define the dependencies (arrows) between the services.

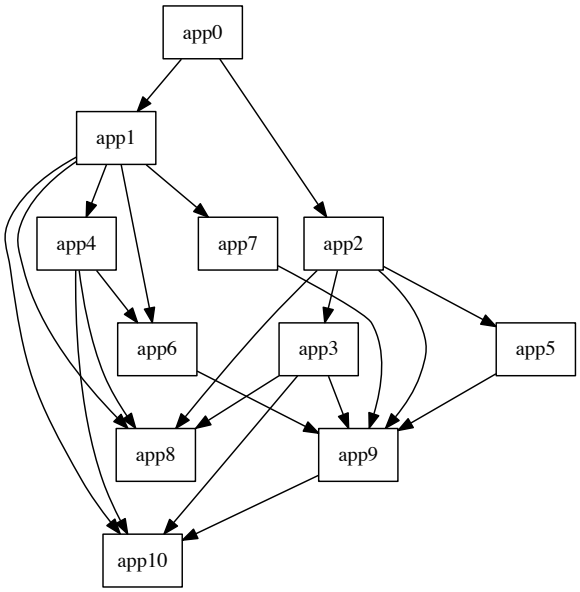


Figure 6.20: **System-C**: a cloud-based system where there are n application services where the dependencies between the services are acyclic and generated randomly. This figure shows an example system with 10 application services, whose dependencies were generated randomly, and 1 front-end application service. The boxes are the services which are running on VMs, and the arrows are the dependencies between the services.

Every service is running on a VM. Thus, the VM must be created before installing and starting the service.

For n services and $n \geq 5$, the script generated 10 random graphs which were used to generate 10 different configurations that have the same number of services but different combination of dependencies. We did not check the isomorphic property between the graphs because the checking process is an NP-hard problem. However, a graph will not be used if it has the same set of edges with previously generated graphs. Figure 6.20 shows an example system with 11 application services where `app0` is the front-end service of the system. §C.3.1.1 and §C.3.1.2 show configuration tasks for the cloud-deployment scenario this example system in SFP and PDDL respectively. While the configuration tasks for cloud-burst scenario are shown in §C.3.2.1 and §C.3.2.2.

6.3.3 Results and Analysis

The followings are the experiment results and analysis of all use-cases.

6.3.3.1 System-A

For each scenario, we conducted two experiments using system-A. In the first experiment, we generated 100 configurations where m and n are ranging from 1 to 10, and the number of services is $(m \times n) + 1$ for the cloud deployment scenario and $((m \times n) + 1) \times 2$ for the cloud burst scenario.

To give a better analysis of the effect of different number of subsystems (m) and different number of layers (n) of system-A to the performance of the planners, we conducted the second experiment using two different datasets of configurations. In the first dataset, we set m to be fixed at 10 and n is set ranging from 1 to 50. On the other hand, the second dataset has configurations where n is fixed at 10 and m is set ranging from 1 to 50. Thus, each dataset has 50 different configurations.

Cloud Deployment Scenario

Tables in figure 6.21 show the results of the first experiment where m and n are ranging from 1 to 10. These tables show that Nuri^{FF} and Nuri^{LM} solved all configuration tasks, while MIPS-XXL can only solve the tasks when the system has less than 80 services – it cannot solve 6 (of 100) tasks due to timeout. The planning time of MIPS-XXL was significantly increasing when the size of the system increases, for example: 129.07 and 231.43 seconds for $\{m = 10, n = 6\}$ and $\{m = 10, n = 7\}$ respectively. On the other hand, there is no significant increase in planning time of Nuri^{FF} (2 and 2.43 seconds) and Nuri^{LM} (2.03 and 2.78 seconds). In general, Nuri^{FF} has a slightly better planning time (about 1 second) than Nuri^{LM} . And for the largest task $\{m = 10, n = 10\}$ (101 services), both generated a sequential plan with 404 actions.

In the second experiment, figure 6.22 shows the planning times of MIPS-XXL, Nuri^{FF} , and Nuri^{LM} for the first dataset (m is fixed at 10). This figure shows that all problems can be solved by Nuri^{FF} and Nuri^{LM} . Similar to the previous results, MIPS-XXL can only solve any task when the number of services is less than 80. The graph also shows that in general, Nuri^{FF} outperforms Nuri^{LM} for average 7.7 seconds. For the largest task $\{m = 10, n = 50\}$ (501 services), Nuri^{FF} and Nuri^{LM} solved the task in 483.54 seconds and 482.87 seconds respectively. They generated a sequential plan consisting of 2004 actions.

MIPS-XXL

time(s)	**n** = 1	2	3	4	5	6	7	8	9	10
m = 1	0.03	0.05	0.06	0.07	0.09	0.11	0.15	0.18	0.22	0.27
2	0.05	0.07	0.12	0.18	0.27	0.43	0.65	0.99	1.5	2.19
3	0.06	0.11	0.21	0.44	0.83	1.59	2.55	3.93	5.93	8.68
4	0.08	0.2	0.44	1.04	2.22	3.92	6.74	11.22	17.07	25.93
5	0.09	0.27	0.88	2.4	4.81	8.82	15.63	25.78	40.4	61.46
6	0.12	0.44	1.6	3.99	8.78	18.21	31.45	53.65	82.94	125.89
7	0.15	0.67	2.72	7.03	16.03	31.45	57.34	96.89	153.12	231.22
8	0.18	1.1	4.13	11.95	28.14	53.2	97.7	162.44	256.62	to
9	0.23	1.63	6.28	17.81	42.57	87.23	161.2	266.73	to	to
10	0.3	2.18	9.54	27.83	62.71	129.07	231.43	to	to	to

Nuri^{FF}

time(s)	**n** = 1	2	3	4	5	6	7	8	9	10
m = 1	0.25	0.26	0.29	0.29	0.29	0.32	0.35	0.34	0.34	0.41
2	0.28	0.29	0.3	0.35	0.39	0.4	0.42	0.5	0.52	0.6
3	0.26	0.31	0.37	0.4	0.47	0.51	0.52	0.59	0.65	0.82
4	0.28	0.35	0.4	0.44	0.51	0.58	0.72	0.74	0.91	1.02
5	0.28	0.34	0.44	0.57	0.6	0.76	0.85	0.94	1.15	1.34
6	0.33	0.39	0.52	0.6	0.69	0.85	1.05	1.32	1.49	1.86
7	0.35	0.44	0.5	0.66	0.82	1.1	1.26	1.74	2.09	2.42
8	0.32	0.44	0.58	0.8	0.94	1.24	1.69	2.05	2.43	3.45
9	0.35	0.52	0.63	0.85	1.13	1.57	1.99	2.41	3.21	4.57
10	0.39	0.53	0.71	0.91	1.45	2.0	2.43	3.08	4.08	5.21

Nuri^{LM}

time(s)	**n** = 1	2	3	4	5	6	7	8	9	10
m = 1	0.24	0.28	0.3	0.29	0.29	0.31	0.34	0.34	0.36	0.38
2	0.28	0.27	0.3	0.34	0.41	0.4	0.44	0.5	0.52	0.54
3	0.27	0.31	0.37	0.4	0.45	0.53	0.57	0.58	0.67	0.82
4	0.31	0.34	0.4	0.5	0.55	0.59	0.7	0.8	0.99	1.1
5	0.29	0.38	0.47	0.51	0.62	0.75	0.95	1.08	1.29	1.47
6	0.31	0.39	0.49	0.61	0.79	0.91	1.14	1.5	1.76	2.1
7	0.31	0.43	0.59	0.69	0.87	1.15	1.54	1.85	2.19	2.82
8	0.34	0.5	0.6	0.79	1.06	1.47	1.85	2.24	3.02	4.05
9	0.37	0.47	0.67	0.9	1.32	1.71	2.17	3.02	3.86	4.72
10	0.36	0.52	0.8	1.07	1.43	2.03	2.78	4.01	4.72	6.29

Figure 6.21: The tables show the planning time for system-A in the cloud deployment scenario. From top to bottom are the planning times of MIPS-XXL, Nuri^{FF} and Nuri^{LM}. Note that “to” equals to **timeout** – the planner cannot find the solution within the deadline.

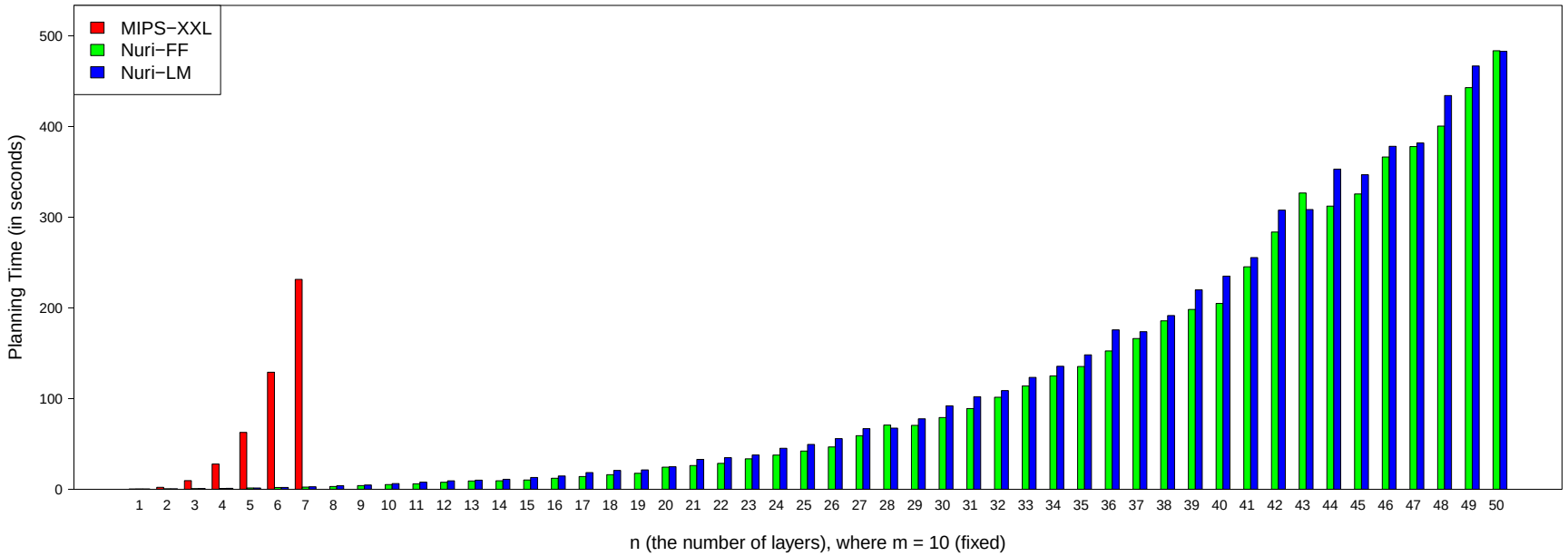


Figure 6.22: The planning time (y-axis) for system-A in the cloud deployment scenario where m is fixed at 10 and n (x-axis) is ranging from 1 to 50. Note that the number of services is $(n \times m) + 1$. From 50 tasks, Nuri^{FF} and Nuri^{LM} solved all tasks, while MIPS-XXL only solved 7 tasks.

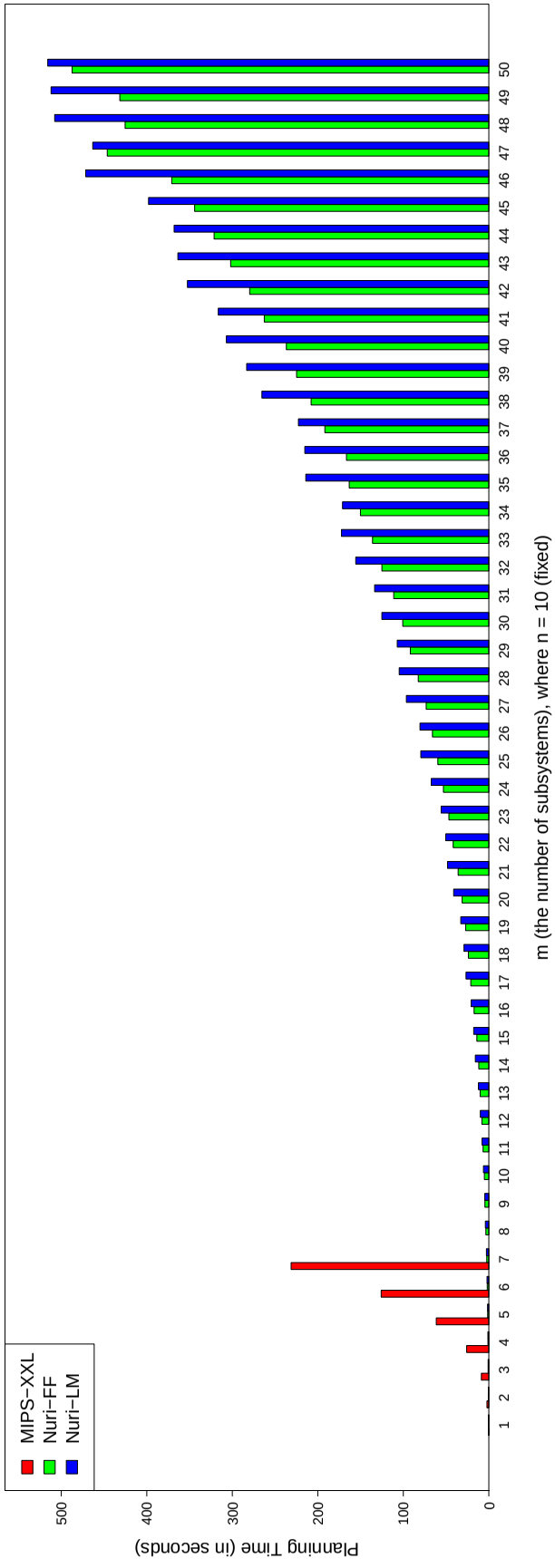


Figure 6.23: The planning time (y-axis) for system-A in the cloud deployment scenario where m (x-axis) is ranging from 1 to 50 and n is fixed at 10. Note that the number of services is $(n \times m) + 1$. From 50 tasks, Nuri^{FF} and Nuri^{LM} solved all tasks, while MIPS-XXL only solved 7 tasks.

515.81 seconds respectively.

By comparing figure 6.22 and 6.23, we may notice that Nuri^{FF} and Nuri^{LM} have slightly worse planning time on the second dataset compared to the first dataset.

From the tables and the figures, notice that Nuri^{FF} significantly outperforms MIPS-XXL. Since MIPS-XXL is using h^{FF} heuristic in search, which is the same as Nuri^{FF} , then this indicates that our compilation technique for handling the global constraints (in particular for the simple implication) is better than the technique used in MIPS-XXL. §6.3.4 discusses this in more details.

Cloud Burst Scenario

The tables in figure 6.24 show the results of the first experiment: the planning times of MIPS-XXL, Nuri^{FF} and Nuri^{LM} where m and n are ranging from 1 to 10. These tables show that Nuri^{LM} solved all configuration tasks, Nuri^{FF} can only solve 42 (of 100) tasks, while MIPS-XXL can only solve 23 (of 100) tasks. Nuri^{FF} and MIPS-XXL cannot solve other problems because they exceeded the memory limit ("mem") and timeout ("to") respectively. Note that for the largest task $\{m = 10, n = 10\}$ (202 services), Nuri^{LM} generated a sequential plan with 911 steps within 275.75 seconds.

Figure 6.24 show that Nuri^{LM} significantly outperforms Nuri^{FF} , which is opposite to the results of the cloud deployment scenario (see figure 6.21) where Nuri^{FF} slightly outperforms Nuri^{LM} . We suspect that this is because the current and the desired state of every task of the cloud burst scenario have very small differences – they only differ in the location of the virtual machines of the main system. When h^{FF} (used by Nuri^{FF}) generates the relaxed planning graph to compute the heuristic values, the goal state can be reached in only a few steps. Thus, the solution of the relaxed planning problem has less number of actions compared to the original solution. Some actions which are required in the solution plan of the original problem, such as starting services after stopping them, have heuristic value equal to infinity since they are not part of the solution plan of the relaxed planning problem. This makes the search engine visiting a large number of superfluous states, which significantly effects the overall planning time. §6.3.4 discusses this in more details.

On the other hand, Nuri^{LM} computes the heuristic values based on the landmarks. It starts from the known landmarks which are the goal atoms, and then uses the causal-links defined in actions' precondition/effect in order to find other landmarks. Thus, the required adversary actions will be set with better heuristic value since their effects are parts of the landmarks.

MIPS-XXL

time(s)	**n** = 1	2	3	4	5	6	7	8	9	10
m = 1	0.02	0.06	0.13	0.26	0.54	1.07	2.16	3.53	5.58	8.56
2	0.06	0.61	3.94	16.43	52.64	141.98	to	to	to	to
3	0.29	12.36	156.4	to	to	to	to	to	to	to
4	1.84	98.62	to	to	to	to	to	to	to	to
5	10.63	to	to	to	to	to	to	to	to	to
6	70.92	to	to	to	to	to	to	to	to	to
7	to	to	to	to	to	to	to	to	to	to
8	to	to	to	to	to	to	to	to	to	to
9	to	to	to	to	to	to	to	to	to	to
10	to	to	to	to	to	to	to	to	to	to

Nuri^{FF}

time(s)	**n** = 1	2	3	4	5	6	7	8	9	10
m = 1	0.32	0.34	0.32	0.35	0.44	0.44	0.54	0.72	1.4	3.4
2	0.33	0.38	0.43	0.61	1.36	7.11	53.82	426.34	mem	mem
3	0.35	0.39	0.65	3.48	53.37	984.95	mem	mem	mem	mem
4	0.41	0.55	2.49	80.51	mem	mem	mem	mem	mem	mem
5	0.42	0.75	18.14	mem	mem	mem	mem	mem	mem	mem
6	0.41	1.39	171.57	mem	mem	mem	mem	mem	mem	mem
7	0.44	4.96	mem	mem	mem	mem	mem	mem	mem	mem
8	0.44	19.9	mem	mem	mem	mem	mem	mem	mem	mem
9	0.51	94.3	mem	mem	mem	mem	mem	mem	mem	mem
10	0.6	412.72	mem	mem	mem	mem	mem	mem	mem	mem

Nuri^{LM}

time(s)	**n** = 1	2	3	4	5	6	7	8	9	10
m = 1	0.27	0.3	0.59	0.32	0.36	0.4	0.46	0.46	0.53	0.6
2	0.28	0.37	0.45	0.44	0.59	0.64	0.74	0.88	1.02	1.16
3	0.34	0.42	0.54	0.65	0.8	1.06	1.37	1.76	2.37	3.01
4	0.35	0.54	0.7	0.99	1.38	1.83	2.66	3.75	5.24	7.48
5	0.4	0.57	0.86	1.21	1.86	3.0	4.95	7.65	11.78	16.68
6	0.41	0.6	1.01	1.81	3.17	5.67	9.44	15.65	23.63	36.84
7	0.44	0.74	1.36	2.68	5.31	10.21	17.19	28.89	42.41	62.98
8	0.47	0.96	2.0	4.11	8.51	15.72	28.22	48.43	73.65	109.77
9	0.48	1.05	2.65	5.94	12.61	24.99	43.99	73.91	116.85	175.97
10	0.56	1.37	3.59	8.75	18.97	37.55	67.57	114.68	178.71	275.75

Figure 6.24: The tables show the planning time for system-A in the cloud burst scenario. From top to bottom are the planning times of MIPS-XXL, Nuri^{FF} and Nuri^{LM}. Note that “to” means **timeout**, and “mem” means **out of memory**

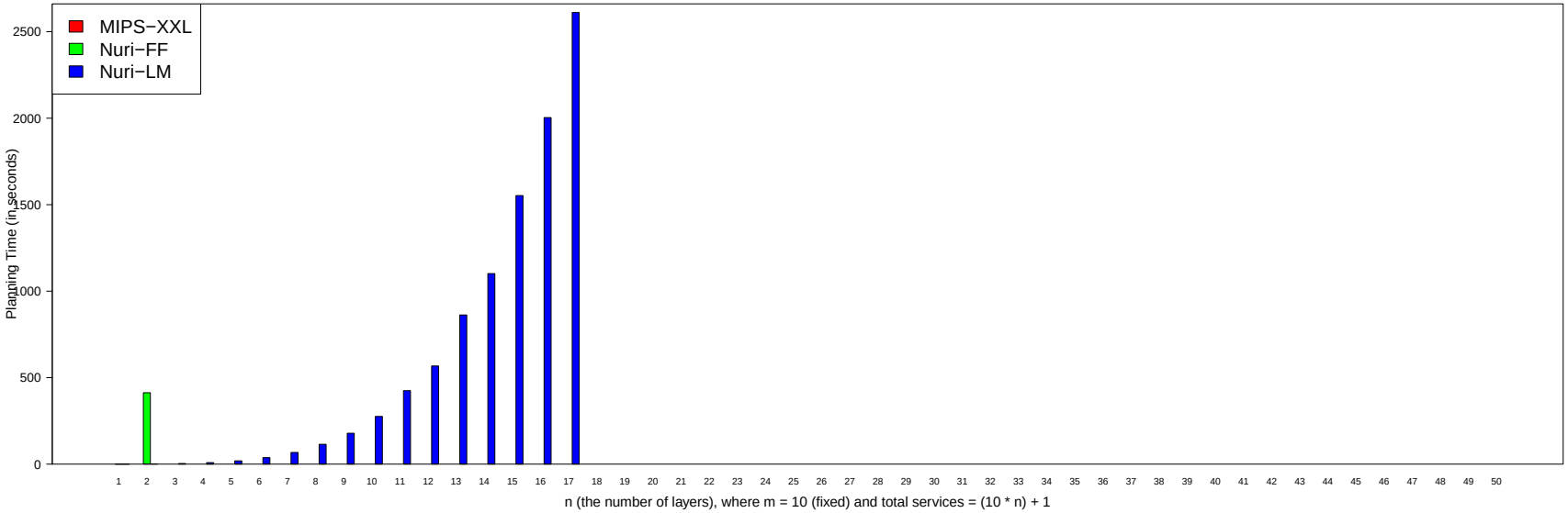


Figure 6.25: The planning time (y-axis) for system-A in the cloud burst scenario where m is fixed at 10 and n (x-axis) is ranging from 1 to 50. Note that the number of services is $(n \times m) + 1$. From 50 tasks, Nuri^{LM} solved 17 tasks, Nuri^{FF} solved 2 tasks, and MIPS-XXL solved 1 task.

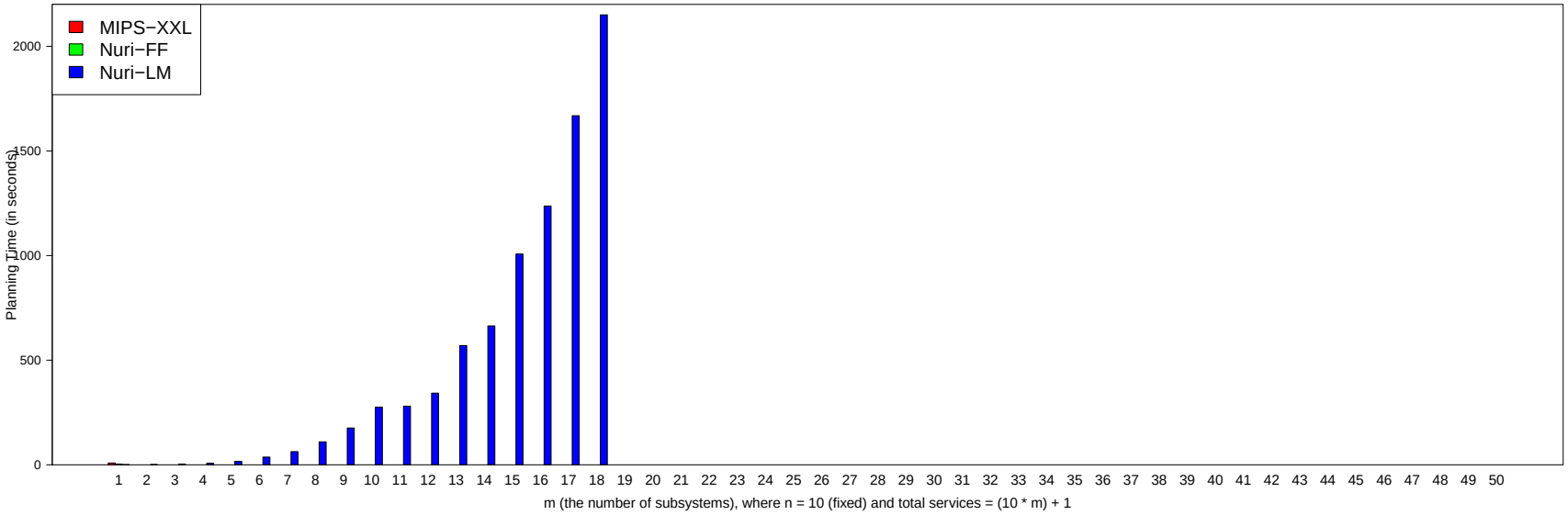


Figure 6.26: The planning time (y-axis) for system-A in the cloud burst scenario where m (x-axis) is ranging from 1 to 50 and n is fixed at 10. Note that the number of services is $(n \times m) + 1$. From 50 tasks, Nuri^{LM} solved 18 tasks, Nuri^{FF} solved 1 task, and MIPS-XXL solved none.

seconds, and the plan consists of 1541 actions.

Figure 6.26 shows the planning times of MIPS-XXL, Nuri^{FF} , and Nuri^{LM} for the second dataset (n is fixed at 10). The figure shows that Nuri^{LM} solved 18 (of 50) tasks, Nuri^{FF} solved 1 (of 50) task, and MIPS-XXL cannot solve any task (always timeout). Nuri^{LM} and Nuri^{FF} cannot solve other task because out of memory. The largest task that can be solved by Nuri^{LM} is $\{m = 18, n = 10\}$ (362 services), which is solved in 2150.34 seconds, and the plan consists of 1631 actions.

Similar to the results of the cloud deployment scenario, Nuri^{LM} suffers degradation of performance when dealing with the second dataset (17 solved tasks) comparing to the first one (18 solved tasks). On the other hand, although Nuri^{FF} cannot solve all tasks, but it significantly outperforms MIPS-XXL. This confirms the previous claim that our compilation technique for handling the global constraints (in particular for the simple implication) is better than the technique used in MIPS-XXL.

6.3.3.2 System-B

For each scenario, we conducted two experiments using system-B. In the first experiment, we generated 100 configurations where m and n are ranging from 1 to 10, and the number of services is $(m \times n) + n$ and $((m \times n) + n) \times 2$ for the cloud deployment and the cloud burst scenarios respectively.

To give a better analysis of the effect of different number of subsystems (m) and different number of layers (n) to the performance of the planners, we conducted the second experiment using two different datasets of configurations: m to be fixed at 10 and n is set ranging from 1 to 50; n is fixed at 10 and m is set ranging from 1 to 50. Notice that each dataset has 50 different configurations.

Cloud Deployment Scenario

The tables in figure 6.27 show the results of the first experiment where m and n are ranging from 1 to 10. These tables show that Nuri^{FF} and Nuri^{LM} solved all configuration tasks, while MIPS-XXL solved 86 (of 100) tasks – other 14 tasks cannot be solved due to timeout. The planning time of MIPS-XXL was significantly increasing when the size of the system increases, for example: 133.25 and 277.43 seconds for $\{m = 10, n = 5\}$ and $\{m = 10, n = 6\}$ respectively. On the other hand, there is no significant increase in the planning time of Nuri^{FF} (2.13 and 3.03 seconds) and Nuri^{LM} (2.97 and 3.87 seconds). In general, Nuri^{FF} has a slightly better planning time (about 1 seconds) than Nuri^{LM} . For the largest task $\{m = 10, n = 10\}$ (110 services), both generated a sequential plan with 440 actions.

The results of the second experiment using the first dataset (m is fixed at 10) are depicted in figure 6.28. The figure shows that Nuri^{FF} and Nuri^{LM} solved all tasks, while MIPS-XXL only solved 6 (of 50) tasks – others cannot be solved due to timeout. The figure also shows that Nuri^{LM} outperforms Nuri^{FF} for about (average) 111.62 seconds. Nuri^{FF} and Nuri^{LM} solved the largest task $\{m = 10, n = 50\}$ (550 services) in 1529.54 and 842.27 seconds respectively. They generated a sequential plan consisting of 2200 actions.

Figure 6.29 shows the experiment results using the second dataset (n is fixed at 10). The figure shows that MIPS-XXL only solved 5 (of 50) tasks, while Nuri^{FF} and Nuri^{LM} solved all tasks. Similar above, Nuri^{LM} outperforms Nuri^{FF} for about (average) 61.89 seconds. The largest task $\{m = 50, n = 10\}$ (510 services) was solved by Nuri^{FF} and Nuri^{LM} in 1641.65 and 1252.30 seconds respectively. They generated

MIPS-XXL

time(s)	**n** = 1	2	3	4	5	6	7	8	9	10
m = 1	0.04	0.06	0.1	0.14	0.26	0.34	0.53	0.81	1.23	1.77
2	0.04	0.09	0.19	0.39	0.77	1.45	2.6	4.05	6.16	9.11
3	0.06	0.16	0.43	1.05	2.27	4.29	7.59	12.89	20.38	30.53
4	0.08	0.18	0.74	2.03	4.9	10.81	19.88	32.5	52.23	79.32
5	0.05	0.35	1.39	4.31	10.84	22.2	41.06	70.36	112.61	171.85
6	0.06	0.59	2.63	8.22	20.16	40.97	76.85	133.92	213.57	to
7	0.15	1.04	4.65	14.91	36.17	73.47	136.1	230.61	to	to
8	0.18	1.64	7.26	22.28	55.6	122.02	219.85	to	to	to
9	0.23	2.14	10.72	33.64	84.54	196.06	to	to	to	to
10	0.21	3.06	16.09	49.89	133.25	277.43	to	to	to	to

Nuri^{FF}

time(s)	**n** = 1	2	3	4	5	6	7	8	9	10
m = 1	0.39	0.37	0.37	0.48	0.36	0.48	0.41	0.55	0.48	0.79
2	0.27	0.36	0.42	0.38	0.43	0.59	0.54	0.61	0.78	0.9
3	0.28	0.32	0.39	0.46	0.52	0.61	0.7	0.8	0.93	1.28
4	0.28	0.36	0.44	0.53	0.61	0.78	0.91	1.09	1.3	1.76
5	0.29	0.38	0.49	0.59	0.79	0.94	1.18	1.48	1.77	2.51
6	0.31	0.42	0.54	0.68	0.94	1.19	1.52	1.92	2.4	3.4
7	0.32	0.45	0.61	0.82	1.08	1.71	2.07	2.62	3.31	4.65
8	0.33	0.48	0.75	1.02	1.38	1.84	2.44	3.46	4.24	6.05
9	0.37	0.54	0.76	1.1	1.69	2.26	3.0	4.39	5.96	7.97
10	0.41	0.65	1.08	1.44	2.13	3.03	4.25	5.67	7.55	9.62

Nuri^{LM}

time(s)	**n** = 1	2	3	4	5	6	7	8	9	10
m = 1	0.42	0.42	0.35	0.4	0.44	0.43	0.48	0.61	0.59	1.14
2	0.37	0.48	0.42	0.58	0.63	0.55	0.73	0.85	0.93	1.44
3	0.47	0.52	0.57	0.65	0.75	0.81	0.92	0.96	1.14	1.64
4	0.29	0.38	0.44	0.55	0.9	0.86	1.05	1.3	1.62	2.19
5	0.33	0.45	0.49	0.74	0.9	1.14	1.44	1.77	2.17	2.99
6	0.32	0.43	0.61	0.74	1.0	1.4	1.84	2.29	2.95	3.71
7	0.33	0.51	0.74	0.95	1.31	1.67	2.22	2.96	3.66	5.05
8	0.37	0.55	0.77	1.15	1.6	2.01	2.99	4.07	4.83	6.67
9	0.4	0.6	0.89	1.34	1.94	2.66	3.69	4.89	6.5	9.01
10	0.39	0.71	1.65	1.76	2.97	3.87	4.98	6.65	8.52	10.97

Figure 6.27: The tables show the planning time for system-B in the cloud deployment scenario. From top to bottom are the planning times of MIPS-XXL, Nuri^{FF} and Nuri^{LM} ("to" equals to **timeout**).

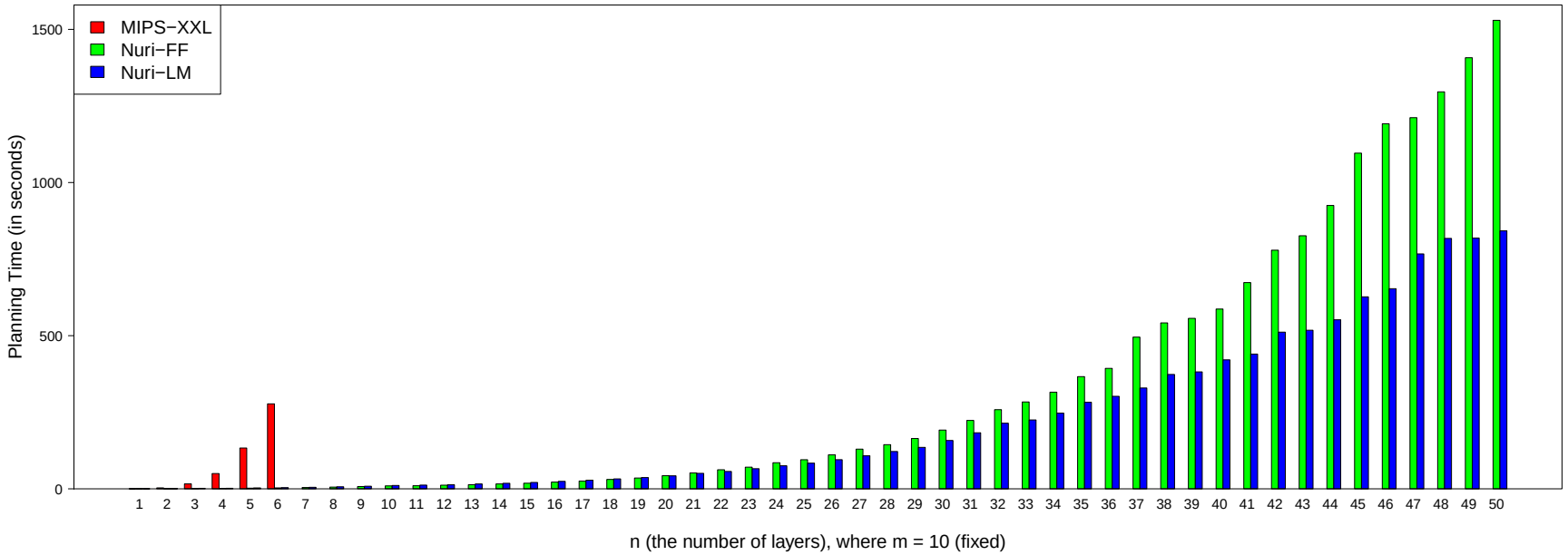


Figure 6.28: The planning time (y-axis) for system-B in the cloud deployment scenario where m is fixed at 10 and n (x-axis) is ranging from 1 to 50. Note that the number of services is $(n \times m) + n$. From 50 tasks, Nuri^{LM} and Nuri^{FF} solved all tasks, while MIPS-XXL only solved 6 task.

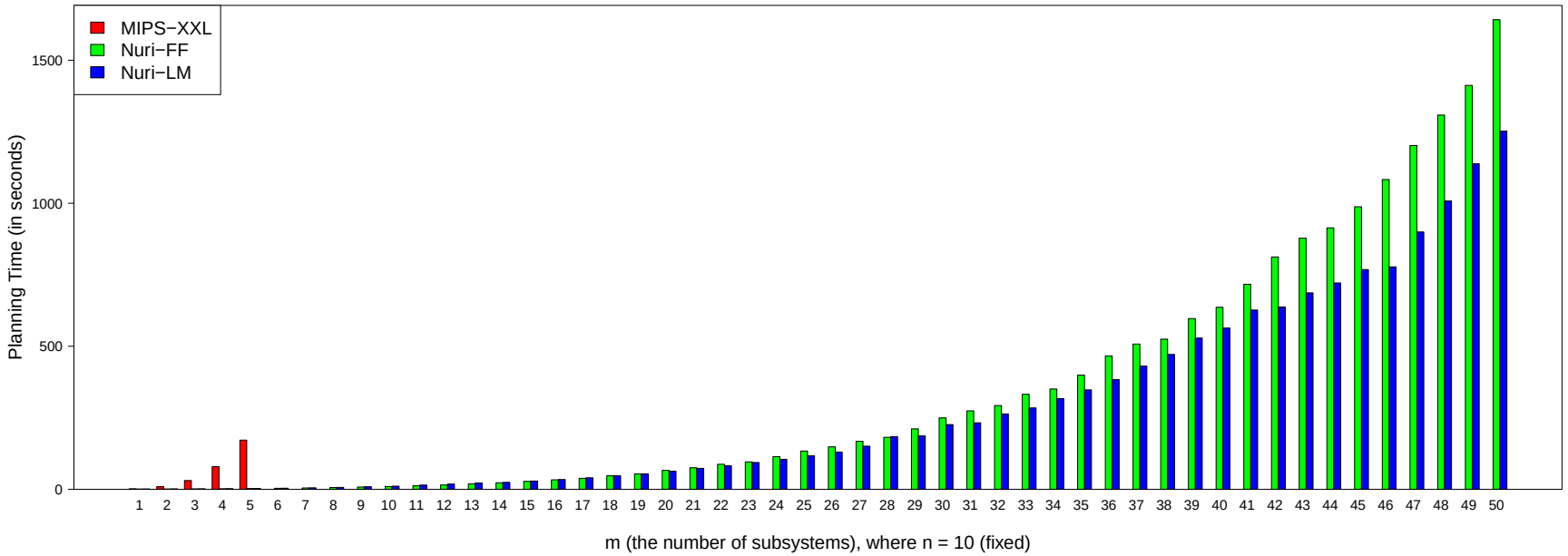


Figure 6.29: The planning time (y-axis) for system-B in the cloud deployment scenario where m (x-axis) is ranging from 1 to 50 and n is fixed at 10. Note that the number of services is $(n \times m) + n$. From 50 tasks, Nuri^{LM} and Nuri^{FF} solved all tasks, while MIPS-XXL solved 5 task.

a sequential plan with 2040 actions.

The tables and the figures show that Nuri^{FF} significantly outperforms MIPS-XXL. This confirms the above claim that our technique for handling global constraints (in particular for the simple implication) is better than MIPS-XXL.

MIPS-XXL

time(s)	**n** = 1	2	3	4	5	6	7	8	9	10
m = 1	0.02	0.05	0.12	0.26	0.56	1.02	2.05	3.57	5.74	8.83
2	0.27	8.59	278.61	to	to	to	to	to	to	to
3	10.27	to	to	to	to	to	to	to	to	to
4	to	to	to	to	to	to	to	to	to	to
5	to	to	to	to	to	to	to	to	to	to
6	to	to	to	to	to	to	to	to	to	to
7	to	to	to	to	to	to	to	to	to	to
8	to	to	to	to	to	to	to	to	to	to
9	to	to	to	to	to	to	to	to	to	to
10	to	to	to	to	to	to	to	to	to	to

Nuri^{FF}

time(s)	**n** = 1	2	3	4	5	6	7	8	9	10
m = 1	0.29	0.3	0.34	0.36	0.35	0.39	0.5	0.71	1.36	3.22
2	0.31	0.34	0.42	0.59	1.51	7.21	47.13	336.83	mem	mem
3	0.38	0.45	0.88	6.61	94.62	mem	mem	mem	mem	mem
4	0.52	0.76	9.83	290.81	mem	mem	mem	mem	mem	mem
5	0.51	3.37	188.6	mem	mem	mem	mem	mem	mem	mem
6	0.66	28.36	mem	mem	mem	mem	mem	mem	mem	mem
7	1.39	278.42	mem	mem	mem	mem	mem	mem	mem	mem
8	4.91	mem	mem	mem	mem	mem	mem	mem	mem	mem
9	19.39	mem	mem	mem	mem	mem	mem	mem	mem	mem
10	86.5	mem	mem	mem	mem	mem	mem	mem	mem	mem

Nuri^{LM}

time(s)	**n** = 1	2	3	4	5	6	7	8	9	10
m = 1	0.26	0.29	0.34	0.34	0.41	0.41	0.48	0.52	0.54	0.58
2	0.3	0.42	0.49	0.54	0.66	0.78	0.97	1.14	1.41	1.72
3	0.41	0.49	0.66	0.85	1.12	1.57	2.13	2.93	3.72	5.11
4	0.43	0.58	0.83	1.3	1.97	3.19	4.73	7.27	10.99	15.7
5	0.51	0.75	1.29	2.29	3.94	6.71	10.85	16.96	26.43	39.24
6	0.59	1.02	1.93	4.1	7.42	12.95	22.66	35.65	54.99	80.88
7	0.69	1.38	3.15	6.63	13.3	24.08	40.98	67.23	102.0	153.19
8	0.77	2.08	4.92	10.63	21.47	40.78	70.55	113.75	178.6	264.54
9	1.03	2.83	8.05	18.99	38.88	73.91	114.42	186.32	289.4	428.13
10	1.31	3.82	10.55	24.84	51.79	97.98	182.85	293.79	466.05	698.83

Figure 6.30: The tables show the planning time for system-B in the cloud burst scenario. From top to bottom are the planning times of MIPS-XXL, Nuri^{FF} and Nuri^{LM} . Note that “to” means **timeout**, and “mem” means **out of memory**.

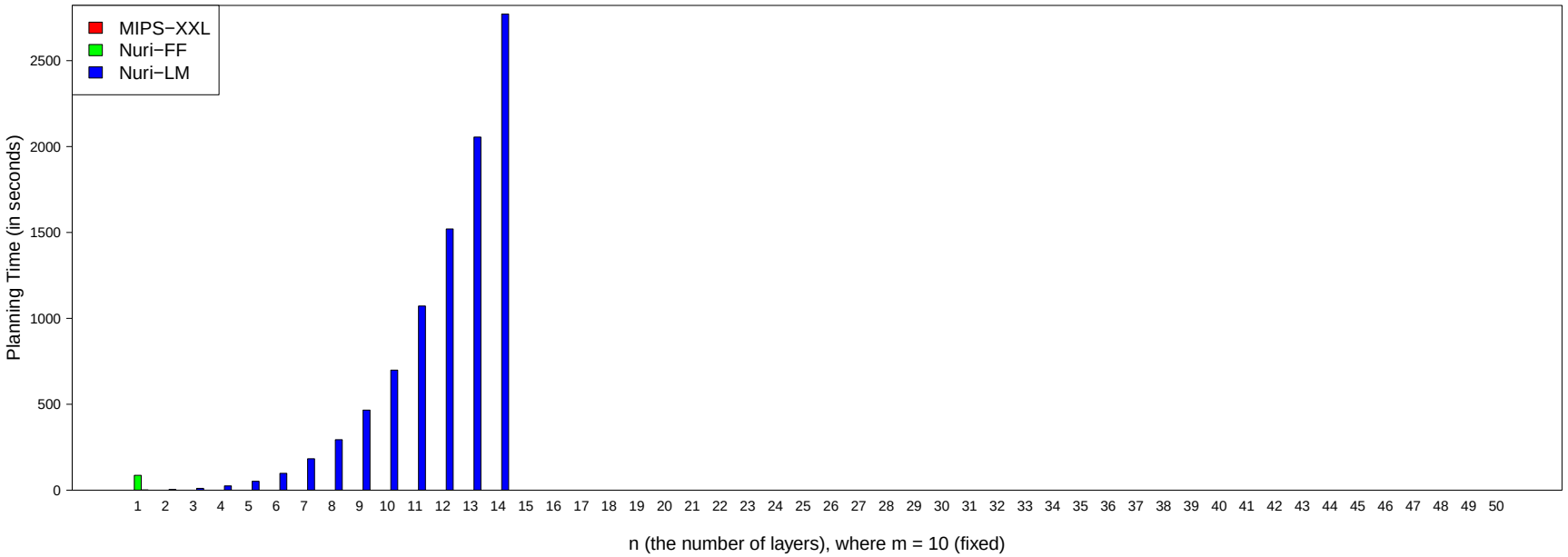


Figure 6.31: The planning time (y-axis) for system-B in the cloud burst scenario where m is fixed at 10 and n (x-axis) is ranging from 1 to 50. Note that the number of services is $(n \times m) + n$. From 50 tasks, Nuri^{LM} solved 14 tasks, while Nuri^{FF} and MIPS-XXL only solved 1 task.

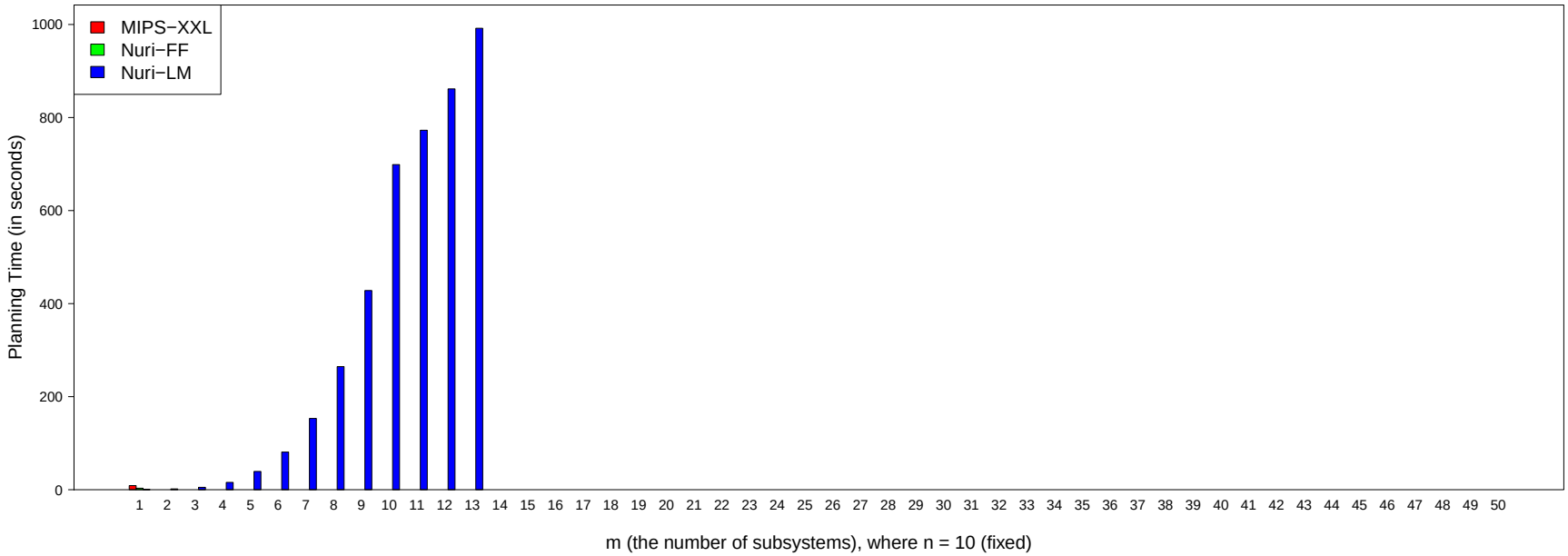


Figure 6.32: The planning time (y-axis) for system-B in the cloud burst scenario where m (x-axis) is ranging from 1 to 50 and n is fixed at 10. Note that the number of services is $(n \times m) + n$. From 50 tasks, Nuri^{LM} solved 13 tasks, Nuri^{FF} solved 1 task, and MIPS-XXL solved none.

Cloud Burst Scenario

The tables in figure 6.30 show the results of the first experiment. They show that Nuri^{LM} solved all configuration tasks, Nuri^{FF} solved 37 (of 100) tasks, while MIPS-XXL only solved 14 (of 100) tasks. Nuri^{FF} and MIPS-XXL cannot solve other problems because they exceeded the memory limit ("mem") and timeout ("to") respectively. Note that for the largest task $\{m = 10, n = 10\}$ (220 services), Nuri^{LM} generated a sequential plan with 992 steps within 698.83 seconds.

The results show the superiority of Nuri^{LM} comparing to MIPS-XXL and Nuri^{FF} , which is similar to the results of system-A in the same scenario. Hence, it is likely that heuristic h^{LM} is more suitable than h^{FF} to be used for solving configuration tasks in the cloud burst scenario, regardless of the type of the system.

In the second experiment, figure 6.31 shows the planning times of MIPS-XXL, Nuri^{FF} , and Nuri^{LM} for the first dataset (m is fixed at 10). The figure shows that Nuri^{LM} solved 14 (of 50) tasks, where Nuri^{FF} and MIPS-XXL only solved 1 (of 50) tasks. Nuri^{LM} and Nuri^{FF} cannot solve other tasks because they exceeded the memory limit (24GB), while MIPS-XXL was timeout. The largest task that can be solved by Nuri^{LM} is $\{m = 10, n = 14\}$ (308 services), which is solved in 2771.57 seconds, and the plan consists of 1352 actions.

Figure 6.32 shows the planning times of MIPS-XXL, Nuri^{FF} , and Nuri^{LM} for the second dataset (n is fixed at 10). The figure shows that Nuri^{LM} solved 13 (of 50) tasks, Nuri^{FF} solved 1 (of 50) task, and MIPS-XXL cannot solve any task (always timeout). Nuri^{LM} and Nuri^{FF} cannot solve other task because out of memory. The largest task that can be solved by Nuri^{LM} is $\{m = 13, n = 10\}$ (280 services), which is solved in 991.85 seconds, and the plan consists of 1289 actions.

6.3.3.3 System-C

For the experiments using system-C, we set the number of services ranging from 5 to 100. For each number of services, 10 random directed acyclic graphs were generated where their nodes and edges represent application services and the service dependencies respectively. Thus, there are in total 960 configurations, which are grouped into 96 datasets (based on their number of services).

Cloud Deployment Scenario

Table 6.1: The total solved tasks and the average planning time for generating the workflows for deploying system-C from scratch using MIPS-XXL, Nuri^{FF} , and Nuri^{LM} . Note that "to" equals to timeout.

Total Services	Total Service Dependencies(avg)	Solved Problem (out of 10)			Avg. Planning Time (s)		
		MIPS-XXL	Nuri^{FF}	Nuri^{LM}	MIPS-XXL	Nuri^{FF}	Nuri^{LM}
5	6.1	10	10	10	0.08	0.32	0.47
6	9.2	10	10	10	0.1	0.34	0.5
7	12.7	10	10	10	0.16	0.35	0.53
8	15.2	10	10	10	0.22	0.38	0.54
9	20.0	10	10	10	0.33	0.4	0.58
10	23.3	10	10	10	0.45	0.45	0.6
11	27.4	10	10	10	0.65	0.46	0.59
12	37.4	10	10	10	0.99	0.49	0.61
13	38.8	10	10	10	1.26	0.51	0.64
14	48.0	10	10	10	1.86	0.52	0.66
15	53.8	10	10	10	2.56	0.56	0.7
16	63.0	10	10	10	3.66	0.63	0.76
17	70.0	10	10	10	4.9	0.63	0.79
18	77.9	10	10	10	6.35	0.7	0.83
19	87.3	10	10	10	8.57	0.74	0.82
20	95.9	10	10	10	11.09	0.72	0.96
21	109.4	10	10	10	14.96	0.78	1.02
22	113.5	10	10	10	17.38	0.83	1.01
23	130.1	10	10	10	23.95	0.89	1.11
24	135.8	10	10	10	28.32	0.92	1.14
25	153.9	10	10	10	38.56	0.99	1.18
26	164.0	10	10	10	46.79	1.08	1.26
27	176.9	10	10	10	57.68	1.03	1.3
28	195.7	10	10	10	82.33	1.16	1.38
29	208.8	10	10	10	93.63	1.2	1.45
30	219.1	10	10	10	107.02	1.32	1.52
31	236.6	10	10	10	131.8	1.3	1.6
32	250.2	10	10	10	158.03	1.45	1.67
33	261.5	10	10	10	186.45	1.46	1.7
34	279.1	10	10	10	220.91	1.59	1.82
35	292.6	10	10	10	263.8	1.69	1.87

Continued on next page

Table6.1 – continued from previous page

Total Services	Total Service Dependencies(avg)	Solved Problem (out of 10)			Avg. Planning Time (s)		
		MIPS-XXL	Nuri^{FF}	Nuri^{LM}	MIPS-XXL	Nuri^{FF}	Nuri^{LM}
36	316.1	2	10	10	277.32	1.65	2.01
37	330.5	0	10	10	to	1.78	2.07
38	351.0	0	10	10	to	1.98	2.16
39	375.6	0	10	10	to	1.93	2.3
40	397.6	0	10	10	to	2.18	2.44
41	414.9	0	10	10	to	2.1	2.58
42	426.7	0	10	10	to	2.35	2.67
43	452.6	0	10	10	to	2.33	2.79
44	478.1	0	10	10	to	2.63	2.94
45	493.3	0	10	10	to	2.45	3.07
46	527.4	0	10	10	to	2.74	3.23
47	535.0	0	10	10	to	2.82	3.3
48	576.9	0	10	10	to	3.09	3.53
49	593.7	0	10	10	to	3.01	3.71
50	612.8	0	10	10	to	3.32	3.86
51	634.5	0	10	10	to	3.3	3.96
52	666.1	0	10	10	to	3.53	4.17
53	694.4	0	10	10	to	3.76	4.33
54	712.9	0	10	10	to	3.76	4.49
55	740.8	0	10	10	to	4.07	4.74
56	778.6	0	10	10	to	4.06	4.92
57	795.3	0	10	10	to	4.84	5.12
58	828.4	0	10	10	to	4.51	5.31
59	864.1	0	10	10	to	5.11	5.49
60	881.3	0	10	10	to	4.89	5.7
61	915.9	0	10	10	to	5.17	6.02
62	945.2	0	10	10	to	5.8	6.26
63	969.5	0	10	10	to	5.46	6.43
64	1005.8	0	10	10	to	6.19	6.7
65	1044.2	0	10	10	to	5.83	6.91
66	1069.2	0	10	10	to	6.68	7.15
67	1112.8	0	10	10	to	6.51	7.49
68	1129.5	0	10	10	to	6.51	7.77
69	1160.5	0	10	10	to	7.39	8.0
70	1218.5	0	10	10	to	7.08	8.37
71	1249.1	0	10	10	to	7.65	8.74
72	1266.2	0	10	10	to	7.57	8.95
73	1312.8	0	10	10	to	8.08	9.37
74	1357.8	0	10	10	to	8.61	9.73
75	1390.2	0	10	10	to	8.67	10.06
76	1426.4	0	10	10	to	8.97	10.21
77	1471.6	0	10	10	to	9.29	10.56
78	1502.4	0	10	10	to	10.24	10.97
79	1549.7	0	10	10	to	10.05	11.33
80	1597.2	0	10	10	to	10.29	11.7
81	1633.7	0	10	10	to	10.81	12.17
82	1661.5	0	10	10	to	11.11	12.51
83	1718.2	0	10	10	to	11.14	12.95
84	1762.9	0	10	10	to	12.05	13.45

Continued on next page

Table6.1 – continued from previous page

Total Services	Total Service Dependencies(avg)	Solved Problem (out of 10)			Avg. Planning Time (s)		
		MIPS-XXL	Nuri^{FF}	Nuri^{LM}	MIPS-XXL	Nuri^{FF}	Nuri^{LM}
85	1792.7	0	10	10	to	12.02	13.79
86	1821.1	0	10	10	to	12.6	14.13
87	1867.3	0	10	10	to	12.89	14.6
88	1905.1	0	10	10	to	13.45	15.02
89	1968.7	0	10	10	to	13.98	15.61
90	2002.1	0	10	10	to	14.09	15.96
91	2071.7	0	10	10	to	15.0	16.64
92	2084.6	0	10	10	to	15.08	17.11
93	2159.2	0	10	10	to	15.85	17.64
94	2210.6	0	10	10	to	16.26	18.16
95	2246.5	0	10	10	to	16.4	18.58
96	2274.6	0	10	10	to	17.27	19.08
97	2329.4	0	10	10	to	17.58	19.66
98	2389.6	0	10	10	to	18.18	20.31
99	2414.1	0	10	10	to	19.12	20.59
100	2495.0	0	10	10	to	19.3	21.5
	879.95	312	960	960	50.33	5.73	6.51

Table 6.1 summarises the number of solved problem and the average planning time for every dataset of cloud-deployment scenario using MIPS-XXL, Nuri^{FF} and Nuri^{LM} . The table shows that MIPS-XXL solved 312 (of 960) tasks – it cannot solve tasks that has 37 or more services due to timeout. On the other hand, Nuri^{FF} and Nuri^{LM} solved all tasks in average 5.73 and 6.51 seconds respectively. Nuri^{FF} and Nuri^{LM} solved the largest tasks (100 services and 2495 service dependencies) in average 19.3 and 21.5 seconds respectively. Note that they generated a sequential plan with 404 actions.

Notice that Nuri^{FF} significantly outperforms MIPS-XXL (both planners are using h^{FF} as the heuristic technique). This confirms the previous statement that our technique for handling the global constraints (in particular for the simple implication) is better than MIPS-XXL.

There is an interesting fact on the planning times of large tasks: although the number of service dependencies of system-C is extremely large compared to system-A and system-B, but it did not significantly affect the performance of Nuri^{FF} and Nuri^{LM} . Note that dependencies between system components are defined as global constraints. Thus, the size of the constraint formula is linear to the number of dependencies.

System	Task	Total Services	Total Service Dependencies	Planning Time (s)	
				Nuri^{FF}	Nuri^{LM}
System-A	$\{m = 10, n = 10\}$	101	100	5.21	6.29
System-B	$\{m = 10, n = 9\}$	100	170	7.55	8.52
System-C	100	100	2495	19.3	21.5

The above table shows the planning times of system-A, B, and C in the cloud deployment scenario. All systems have 100-101 services. System-C (2495) has 25 and 13 times more service dependencies than system-A (100) and B (170) respectively. Nuri^{FF} solved the task of system-C in 19.3 seconds, which is just 3.7 and 2.5 times more than the time required for solving system-A and B respectively. And Nuri^{LM} solved the task of system-C in 21.5 seconds, which is just 3.4 and 2.5 times more than the time required for solving system-A and B respectively.

Cloud Burst Scenario

Table 6.2: The total solved tasks and the average planning time for generating the workflows for system-C in the cloud burst scenario using MIPS-XXL, Nuri^{FF} , and Nuri^{LM} . Note that “to” equals to timeout, and “mem” equals to out of memory.

Total Services	Total Service Dependencies(avg)	Solved Problem (out of 10)			Avg. Planning Time (s)		
		MIPS-XXL	Nuri^{FF}	Nuri^{LM}	MIPS-XXL	Nuri^{FF}	Nuri^{LM}
10	12.2	10	10	10	22.89	0.39	0.4
12	18.4	7	10	10	121.6	0.42	0.43
14	25.4	2	10	10	189.4	0.46	0.47
16	30.4	0	10	10	to	0.52	0.54
18	40.0	0	10	10	to	0.58	0.54
20	46.6	0	10	10	to	0.72	0.59
22	54.8	0	10	10	to	0.99	0.65
24	74.8	0	10	10	to	1.52	0.75
26	77.6	0	10	10	to	2.38	0.82
28	96.0	0	10	10	to	4.48	0.87
30	107.6	0	10	10	to	9.3	0.95
32	126.0	0	10	10	to	17.75	1.08
34	140.0	0	10	10	to	47.6	1.17
36	155.8	0	10	10	to	89.9	1.37
38	174.6	0	10	10	to	196.22	1.47
40	191.8	0	10	10	to	390.83	1.63
42	218.8	0	10	10	to	924.73	1.75
44	227.0	0	4	10	to	1499.04	2.01
46	260.2	0	0	10	to	mem	2.24
48	271.6	0	0	10	to	mem	2.48
50	307.8	0	0	10	to	mem	2.73
52	328.0	0	0	10	to	mem	3.05
54	353.8	0	0	10	to	mem	3.38
56	391.4	0	0	10	to	mem	3.58
58	417.6	0	0	10	to	mem	4.15
60	438.2	0	0	10	to	mem	4.71
62	473.2	0	0	10	to	mem	5.12
64	500.4	0	0	10	to	mem	5.78
66	523.0	0	0	10	to	mem	6.03
68	558.2	0	0	10	to	mem	6.91
70	585.2	0	0	10	to	mem	7.42

Continued on next page

Table6.2 – continued from previous page

Total Services	Total Service Dependencies(avg)	Solved Problem (out of 10)			Avg. Planning Time (s)		
		MIPS-XXL	Nuri^{FF}	Nuri^{LM}	MIPS-XXL	Nuri^{FF}	Nuri^{LM}
72	632.2	0	0	10	to	mem	7.59
74	661.0	0	0	10	to	mem	8.24
76	702.0	0	0	10	to	mem	9.89
78	751.2	0	0	10	to	mem	11.09
80	795.2	0	0	10	to	mem	11.62
82	829.8	0	0	10	to	mem	12.76
84	853.4	0	0	10	to	mem	13.91
86	905.2	0	0	10	to	mem	15.01
88	956.2	0	0	10	to	mem	16.11
90	986.6	0	0	10	to	mem	17.6
92	1054.8	0	0	10	to	mem	19.2
94	1070.0	0	0	10	to	mem	19.66
96	1153.8	0	0	10	to	mem	22.1
98	1187.4	0	0	10	to	mem	23.9
100	1225.6	0	0	10	to	mem	25.69
102	1269.0	0	0	10	to	mem	27.25
104	1332.2	0	0	10	to	mem	30.2
106	1388.8	0	0	10	to	mem	32.84
108	1425.8	0	0	10	to	mem	35.5
110	1481.6	0	0	10	to	mem	36.56
112	1557.2	0	0	10	to	mem	42.01
114	1590.6	0	0	10	to	mem	43.11
116	1656.8	0	0	10	to	mem	44.44
118	1728.2	0	0	10	to	mem	48.44
120	1762.6	0	0	10	to	mem	53.64
122	1831.8	0	0	10	to	mem	57.36
124	1890.4	0	0	10	to	mem	61.22
126	1939.0	0	0	10	to	mem	63.21
128	2011.6	0	0	10	to	mem	68.64
130	2088.4	0	0	10	to	mem	72.23
132	2138.4	0	0	10	to	mem	78.36
134	2225.6	0	0	10	to	mem	78.01
136	2259.0	0	0	10	to	mem	83.57
138	2321.0	0	0	10	to	mem	92.49
140	2437.0	0	0	10	to	mem	96.24
142	2498.2	0	0	10	to	mem	100.74
144	2532.4	0	0	10	to	mem	104.68
146	2625.6	0	0	10	to	mem	119.48
148	2715.6	0	0	10	to	mem	119.44
150	2780.4	0	0	10	to	mem	129.59
152	2852.8	0	0	10	to	mem	134.38
154	2943.2	0	0	10	to	mem	139.24
156	3004.8	0	0	10	to	mem	159.52
158	3099.4	0	0	10	to	mem	157.38
160	3194.4	0	0	10	to	mem	172.71
162	3267.4	0	0	10	to	mem	172.91
164	3323.0	0	0	10	to	mem	187.96
166	3436.4	0	0	10	to	mem	191.04
168	3525.8	0	0	10	to	mem	209.21

Continued on next page

Table6.2 – continued from previous page

Total Services	Total Service Dependencies(avg)	Solved Problem (out of 10)			Avg. Planning Time (s)		
		MIPS-XXL	Nuri^{FF}	Nuri^{LM}	MIPS-XXL	Nuri^{FF}	Nuri^{LM}
170	3585.4	0	0	10	to	mem	216.93
172	3642.2	0	0	10	to	mem	227.99
174	3734.6	0	0	10	to	mem	240.67
176	3810.2	0	0	10	to	mem	226.04
178	3937.4	0	0	10	to	mem	270.6
180	4004.2	0	0	10	to	mem	271.69
182	4143.4	0	0	10	to	mem	274.29
184	4169.2	0	0	10	to	mem	309.59
186	4318.4	0	0	10	to	mem	318.51
188	4421.2	0	0	10	to	mem	333.37
190	4493.0	0	0	10	to	mem	342.75
192	4549.2	0	0	10	to	mem	361.73
194	4658.8	0	0	10	to	mem	346.95
196	4779.2	0	0	10	to	mem	395.85
198	4828.2	0	0	10	to	mem	425.82
200	4990.0	0	0	10	to	mem	427.69
	1759.89	19	174	960	76.79	131.52	88.99

Table 6.2 summarises the number of solved problems and the average planning time for every dataset in the cloud burst scenario. The table shows that MIPS-XXL solved 19 (of 960) tasks (other tasks were timeout), Nuri^{FF} solved 174 (of 960) tasks (others tasks were out of memory), and Nuri^{LM} solved all tasks in average 76.98 seconds. Nuri^{LM} solved the largest tasks (200 services) in average 317.03 seconds, and it generated sequential plans wtih 911 actions.

Notice that although Nuri^{FF} cannot solve all tasks, but it still outperforms MIPS-XXL. This result also confirms that our technique is better than MIPS-XXL for handling the global constraints.

System	Task	Total Services	Total Service Dependencies	Planning Time (s) Nuri^{LM}
System-A	$\{m = 10, n = 10\}$	202	201	275.97
System-B	$\{m = 10, n = 9\}$	200	341	466.05
System-C	200	200	4990	427.69

The above table show the planning times of system-A, B, and C in the cloud burst scenario. All systems have 200-202 services. System-C (4990) has 24.8 and 14.6 times more service dependencies than system-A (201) and B (341) respectively. Nuri^{LM} solved the task of system-C in 427.69 which is less than the time required for solving system-B (466.05 seconds), and higher than system-A (275.97 seconds). From this results, it looks like that the more constraints the problem has, it does not mean that

the problem is more difficult to be solved. For the case of system B and C, it seems that the more constraints the problem has, the more easier it is to be solved.

6.3.4 Discussion

Representing dependencies between system components as global constraints was motivated by the purpose for evaluating the efficiency of the compilation technique described in §4.1, which is implemented in Nuri^{FF} and Nuri^{LM} , compared to other technique such as the one implemented in MIPS-XXL, in particular when the number of dependencies is very large. Each dependency is represented as an implication clause where the formula of the global constraints is a conjunction of implication clauses. Since the number of implication clauses is equal to the number of dependencies, then the size of the formula is equal to the number of dependencies as well.

The experiment results consistently show that Nuri^{FF} and Nuri^{LM} outperforms MIPS-XXL in all tasks. Because Nuri^{FF} and MIPS-XXL are using the same heuristic technique which is h^{FF} , then this proves that our compilation technique described in §4.1 is more efficient than the one implemented in MIPS-XXL.

Referring to §6.2.4, MIPS-XXL can generate an exponential number of new actions after compilation in respect to the size of the formula. We suspect that this is the main reason why MIPS-XXL's performance was degrading when solving problems with a large number of dependencies. For example, MIPS-XXL cannot solve cloud-deployment tasks of system-A and system-B that have more than 60 dependencies because it introduces $> 2^{60}$ new actions.

On the other hand, Nuri^{FF} and Nuri^{LM} did not suffer the same problem. Based on our observation, we found that our compilation technique is very efficient such that it did not introduce any new action after compilation. This is because it applies the simple implication compilation rules (see definition 4.3) which can avoid the introduction of new action. Thus, the numbers of actions before and after compilation are the same for all cloud-deployment and cloud-burst tasks. Hence, the sizes of the search space before and after compilation are the same.

Another interesting result from this experiment is that Nuri^{LM} significantly outperforms Nuri^{FF} on solving cloud-burst tasks. For example, Nuri^{LM} can solve all cloud-burst tasks of system-C while Nuri^{FF} can only solve the tasks with maximum 36 services (see §6.3.3.3). Since both planners are using the same compilation technique, then this shows that h^{LM} is generating a better heuristic compared to h^{FF} for

cloud-burst tasks. A better heuristic helps reducing the search time because it helps the planner to avoid visiting superfluous states. We suspect that since the differences between the initial and goal states are small (they differs only on the location of the virtual machines), then h^{FF} does not include some required actions into the solution of the relaxed problem. Hence, these actions will have heuristic values equal to infinity which does not help the planner in action selection during search. The following example illustrates why h^{LM} is better than h^{FF} for cloud-burst tasks.

Consider a configuration task with two services `s1` and `s2`, each of which is running on virtual machine `vm1` and `vm2` respectively, and `s1` depends on `s2`. The virtual machines will be migrated from one (`c1`) to another cloud (`c2`). The task can be described as follows:

initial-state: `vm1.on = c1`, `vm1.running = true`, `s1.running = true`, `vm2.on = c1`, `vm2.running = true`, `s2.running = true`

goal-state: `vm1.on = c2`, `vm1.running = true`, `s1.running = true`, `vm2.on = c2`, `vm2.running = true`, `s2.running = true`

global-constraints: (`s1.running = true` $\Rightarrow$ `s2.running = true`) $\wedge$ (`s1.running = true` $\Rightarrow$ `vm1.running = true`) $\wedge$ (`s2.running = true` $\Rightarrow$ `vm2.running = true`)

actions:

- `vm1.migrate`, pre: (`vm1.on=c1` $\wedge$ `vm1.running=false`), eff: (`vm1.on=c2`)
- `vm1.start`, pre: (`vm1.running=false`), eff: (`vm1.running=true`)
- `vm1.stop`, pre: (`vm1.running=true`), eff: (`vm1.running=false`)
- `s1.start`, pre: (`s1.running=false`), eff: (`s1.running=true`)
- `s1.stop`, pre: (`s1.running=true`), eff: (`s1.running=false`)
- `vm2.migrate`, pre: (`vm2.on=c1` $\wedge$ `vm2.running=false`), eff: (`vm2.on=c2`)
- `vm2.start`, pre: (`vm2.running=false`), eff: (`vm2.running=true`)
- `vm2.stop`, pre: (`vm2.running=true`), eff: (`vm2.running=false`)
- `s2.start`, pre: (`s2.running=false`), eff: (`s2.running=true`)
- `s2.stop`, pre: (`s2.running=true`), eff: (`s2.running=false`)

Using the rules in definition 4.3, the task after compiling the global constraint is:

initial-state: `vm1.on = c1`, `vm1.running = true`, `s1.running = true`, `vm2.on = c1`, `vm2.running = true`, `s2.running = true`

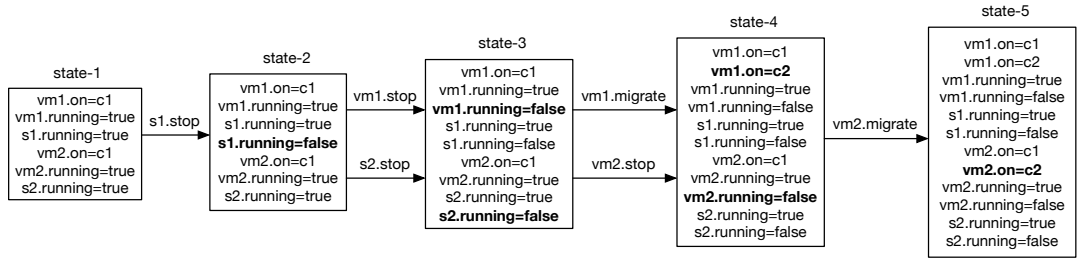
goal-state: `vm1.on = c2`, `vm1.running = true`, `s1.running = true`, `vm2.on = c2`,

vm2.running = true, s2.running = true

actions:

- vm1.migrate, pre: (vm1.on=c1 ∧ vm1.running=false), eff: (vm1.on=c2)
- vm1.start, pre: (vm1.running=false), eff: (vm1.running=true)
- vm1.stop, pre: (vm1.running=true ∧ s1.running=false), eff: (vm1.running=false)
- s1.start, pre: (s1.running=false ∧ vm1.running=true ∧ s2.running=true), eff: (s1.running=true)
- s1.stop, pre: (s1.running=true), eff: (s1.running=false)
- vm2.migrate, pre: (vm2.on=c1 ∧ vm2.running=false), eff: (vm2.on=c2)
- vm2.start, pre: (vm2.running=false), eff: (vm2.running=true)
- vm2.stop, pre: (vm2.running=true ∧ s2.running=false), eff: (vm2.running=false)
- s2.start, pre: (s2.running=false ∧ vm2.running=true), eff: (s2.running=true)
- s2.stop, pre: (s2.running=true ∧ s1.running=false), eff: (s2.running=false)

To solve the above task using h^{FF} , the planner will first compute the heuristic values by generating the following "relaxed" planning graph[11]:



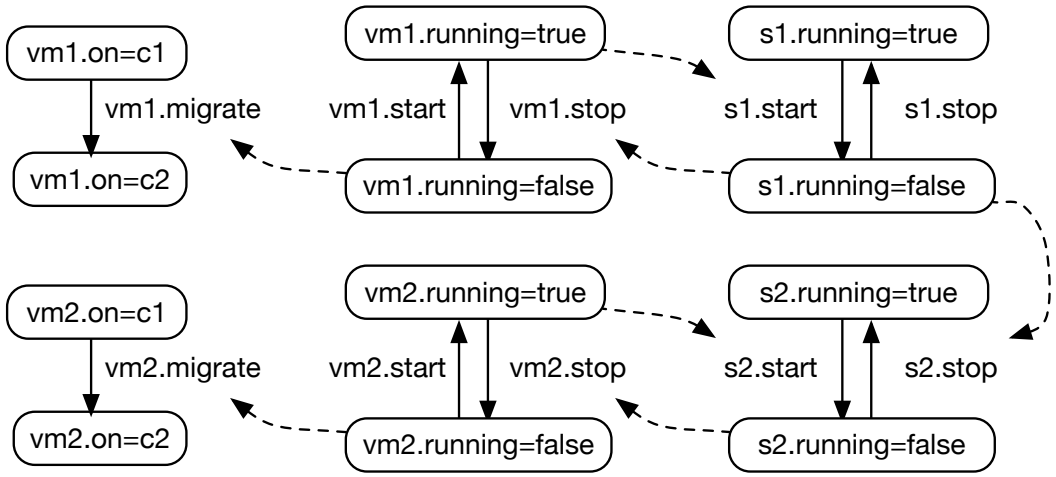
You may notice that the last state-layer contains the goals. Hence, the relaxed planning problem has been solved. Based on the graph, we can set the following heuristic value (distance to the goal) to each action:

Action	Heuristic Value
vm2.migrate	1
vm1.migrate, vm2.stop	2
vm1.stop, s2.stop	3
s1.stop	4
vm1.start, vm2.start, s1.start, s2.start	∞

[11] Based on the principle of relaxed planning problem, a variable can have more than value at particular state (see §2.2.4.1).

From the above table, you may aware that the heuristic value of `vm1.start`, `vm2.start`, `s1.start` and `s2.start` is infinity since they are not part of the solution of the relaxed problem. Because of this, the search engine can select a wrong action at a particular state e.g. selecting `s1.start` rather than `s2.start` first because they have the same heuristic values. This can lead the planner visiting superfluous state. This situation can be found in all cloud-burst tasks. When such situation occurs, then the planner will have to backtrack to find another path. For a large size problem, this backtracking operation can be costly and significantly increase the overall planning time since the total number of possible orderings is exponential in respect to the number of possible actions.

On the other hand, if we set the planner to use h^{LM} as the heuristic technique, then it will first generate the following Domain Transition Graphs[12](DTGs) [Richter et al., 2008] for all variables:



The above DTGs are then used to determine a set of landmarks that should be achieved by every possible plan. Based on definition of landmark (see §2.2.4.2), all goals are landmarks. Hence, `vm1.on = c2`, `vm1.running = true`, `s1.running = true`, `vm2.on = c2`, `vm2.running = true`, `s2.running = true` are landmarks because they are goals of the task. Using the above DTGs, the planner can automatically infer other landmarks:

- Because there is only one transition from `vm1.on = c1` to `vm1.on = c2`, then the preconditions of `vm1.migrate` is landmark.
- `vm1.running = false` is a landmark because it requires by `vm1.migrate` to achieve landmark `vm1.on = c2`. The same thing is also applied to `vm2.running = false`.

[12]In DTG, a node is a possible value that can be assigned to a variable, and an arrow is the transition whose label is the name of the action. More details about DTG can be found in [Helmert, 2006].

- `s1.running = false` is a landmark because it requires by `vm1.stop` to achieve landmark `vm1.running = false`. We use a similar reason to determine `s2.running = false` as a landmark.

The landmarks that are found by h^{LM} are:
{ `vm1.on = c2`, `vm1.running = true`, `vm1.running = false`, `s1.running = true`, `s1.running = false`, `vm2.on = c2`, `vm2.running = true`, `vm2.running = false`, `s2.running = true`, `s2.running = false` }
Based on the effects of the actions and the formula to calculate the heuristics (see §2.2.4.2), these landmarks gives a particular heuristic value to every required actions, in particular the actions where h^{FF} sets infinity as their heuristic value i.e. `vm1.start`, `vm2.start`, `s1.start` and `s2.start`. In addition, we can generate a partial ordering constraints between the landmarks based on the above DTGs e.g. `vm2.running = true` $\prec$ `s2.running = false`. These ordering constraints can help the planner to prioritize an action if its effect achieves a landmark that preceeds landmarks provided by other actions. Thus, h^{LM} gives a better heuristic value compared to h^{FF} , in particular for cloud-burst tasks.

6.3.5 Summary

To summarise the above experiments:

- The results of Nuri^{FF} and Nuri^{LM} validate that our technique described in §4.2 can automatically generate workflows as the solutions of configuration tasks.

- For the cloud deployment scenario, in general, Nuri^{FF} slightly outperforms Nuri^{LM} on tasks of system-A and system-C. Nuri^{FF} also slightly outperforms Nuri^{LM} on small-size tasks (number of services $\leq$ 100) of system-B. However, Nuri^{LM} outperforms Nuri^{FF} on large tasks (number of services > 100) of system-B.

- Nuri^{FF} outperforms MIPS-XXL on all tasks. Since both planners are using the same heuristic that is h^{FF} , then this implies that the compilation technique described in §4.1 is more efficient than the one implemented in MIPS-XXL.

- Nuri^{LM} is superior to Nuri^{FF} on all cloud-burst tasks. Because both planners are using the same compilation technique, then this implies than h^{LM} generates better heuristic values compared to h^{FF} for cloud-burst tasks.

6.4 Planning and Deploying Configuration Changes in Practice

The previous section has shown that the Nuri planner has a promising performance for solving configuration tasks of artificial systems. This section takes us to the next step: it describes experiments which aim to validate that the technique described in §4.2 can be used in practice to generate workflows for configuring real systems. In addition, the experiments also aim to validate that the choreography technique described in §5.2 can be used to deploy the configuration changes.

For this purpose, we have built a prototype configuration tool on top of the Nuri planner as a proof of concept. The tool was then used to configure three real systems in a cloud environment. The tool implements two deployment approaches:

1. Orchestration – the tool automatically generates the workflow (planning), and then a central controller orchestrates the workflow execution;

2. Choreography – the tool automatically choreographs the Behavioural Signature models (choreographing), and then the models were pushed to the agents to enable the distributed execution.

Thus, we measure four things during experiments: the planning time, the choreographing time, the centralised execution time, and distributed execution time.

The next subsection describes the details of the Nuri configuration tool. This is followed by the descriptions, the experiment results and the analysis of three real use-cases.

6.4.1 Nuri

Nuri is a configuration tool that uses SFP as the language to define a declarative configuration specification of the target system. It implements a semi-distributed architecture which consists of a Nuri master and a set of Nuri agents, each of which is controlling a machine. The communications between the master and the agents or between the agents are using HTTP/JSON protocol. All parts of Nuri are implemented in Ruby, except the planner's search engine which is implemented in C++.

The planner is part of the Nuri master. It generates the workflow based on the specification defined by the administrator, the description of the resource components, and the current state of the managed system sent by the agents. If we activate the

choreography mode, then the workflow will be further processed to choreograph the Behavioural Signature (BSig) models.

When Nuri is in *orchestration mode*, then during execution, the orchestrator component of Nuri master sends the actions' description that should be executed by the agents within particular ordering constraints as specified in the workflow. The description consists of the action's name, parameters, preconditions, and effects. The agent will use the preconditions and the effects to assert whether the environment has met the constraints before execution, and assert the execution results. If one of these conditions was not met, either a precondition is not satisfied before execution or an effect is not achieved after execution, then the agent will send a failure response to the master that will terminate the whole execution – this could also trigger re-planning to generate an alternative workflow. Note that the execution manager implements a partial order workflow execution algorithm to exploit the partial-order workflow in order to decrease the execution time.

On the other hand, when *choreography mode* is active, then the master will generate a workflow, choreograph BSig models from the workflow, and then push the models to target agents. Afterwards, the agents will execute the models independently without any control from the master. When an agent requires a remote precondition, then it directly sends a request to the corresponding agent through peer-to-peer communications. Thus, the master has a *loose* coupling with the agents, which is an opposite of the *orchestration mode*.

The Nuri agent implements an architecture shown in figure 5.2. The agent daemon is responsible for:

- Accepting a new model of the machine from the Nuri master;
- Instantiating and managing required resource components based on the model;
- Aggregating the current state of all components to be sent to the Nuri master;
- Executing components' actions based on requests received from the master (*orchestration mode*);
- Receiving a BSig model from the master (*choreography mode*).

All instances of the resource component are organised in a tree structure which is equivalent to the tree structure of SFP objects. An agent can directly query the state of other agents using peer-to-peer communication protocol – this is very useful for sharing configuration between components of different agents, for example: the database component shares its IP address and port to the webservice component where both

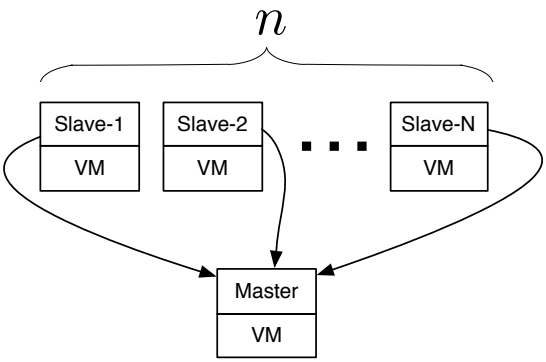


Figure 6.33: A typical Hadoop Cluster with 1 Hadoop master and N slaves.

components are on different machines. The agent also implements the cooperative reactive regression (CRR) (see §5.2.3) algorithm to execute the BSig model.

Every resource component is an instance of a Nuri module. It is responsible for managing a particular resource, for example: a "package" module can be used to manage a software package. A Nuri module is an abstract component which mainly consists of two files: an SFP file that contains a schema (an abstract description of the resource); and a Ruby file that contains a Ruby class as the implementation of the schema. The agent invokes method `update_state` of the components in order to generate the current state of the machine. The agent will also invoke a particular Ruby method of the component in order to execute a particular action as requested by the execution manager.

There is a clear separation between the declarative description in SFP and its Ruby implementation – this allows us to have different implementations, for example one in Ruby and another in Java, communicating transparently using the same configuration language. The mapping from SFP to Ruby and vice versa is done by the agent's daemon using Ruby-SFP library.

The development of Nuri was mainly motivated as a proof of concept of the techniques described in this thesis. It has not yet implemented any particular security model in order to secure the entire system from any malicious attacks. However, it is possible to improve the Nuri's security, for example: using Transport Layer Security (TLS) for securing the communications. Some implementation codes have not been optimised.

6.4.2 Use Case 1: Apache Hadoop

Apache Hadoop is a system that allows for the distributed processing of large data sets across clusters of computers [Foundation, 2014a]. It consists of one Hadoop master and a set of Hadoop slaves whose architecture is illustrated in figure 6.33. Every Hadoop master and slave is running on a virtual machine (VM). The slaves are organised in a flat structure where every slave has a dependency (arrow) with the master and there is no dependency between the slaves, which is similar to the architecture of system-B with a single layer.

We created a Nuri module that has a capability to install/uninstall, configure, and start/stop the Hadoop master and slave services on the target machines. This module consists of three SFP schemata, each of which has a corresponding Ruby class, which are:

- HadoopCommon – contains common configurations and actions shared between the master and the slaves;
- HadoopMaster – contains specific configurations for the master;
- HadoopSlave – contains specific configurations and actions for the slave.

The following codes are the specification of HadoopCommon in SFP:

```
1  schema HadoopCommon extends Package {
2    installed = true
3    running = true
4    configured = true
5    name = "hadoop"
6    provider = "tar"
7    version = "2.2.0"
8    source = "http://repository.foo.com/hadoop"
9    home = "/opt/hadoop"
10   user = "hadoop"
11   group = "hadoop"
12   password = "!"
13   java_home = ""
14   data_dir = "/opt/hadoop/data"
15   def install () {
16     condition {
17       this.installed != true
18     }
19     effect {
20       this.installed = true
21       this.running = false
22       this.configured = false
23     }
24   }
25   ... // other actions
```

```
26  }
27  schema HadoopMaster extends HadoopCommon {
28    cluster_name = "hadoopnuri"
29  }
30  schema HadoopSlave extends HadoopCommon {
31    master: HadoopMaster*
32    def install (master: HadoopMaster) {
33      condition {
34            this.installed != true
35            master.parent.created = true // master.parent is a VM
36          }
37          effect {
38            this.installed = true
39            this.running = false
40            this.configured = false
41            this.master = master
42          }
43    }
44  }
```

In this experiment, the Hadoop system was deployed from scratch to a cloud infrastructure. We used HP Cells (see §2.1.5.4) as the target cloud infrastructure. It is a system that provides a virtual infrastructure where each tenant has secure "containers" called as *Cells*, each of which can have arbitrary virtual machines, virtual storage volumes and virtual networks [hpc, 2014]. Thus, we created a Nuri module called as *cells*[13] that can create a *Cell* container and its virtual resources.

The following is an example of specification for deploying a Hadoop cluster with one master and three slaves (this requires 4 VMs).

```
1  include "modules/vm/vm.sfp"
2  include "modules/cell/cell.sfp"
3  include "modules/hadoop/hadoop.sfp"
4  proxy isa Node {
5    sfpAddress = "localhost"
6    hpcell isa Cell {
7      access_uri = "https://cell.foo.com/:8553/caas/rest/v1/cells"
8      name = "herry-cell"
9    }
10 }
11 master isa VM {
12   in_cloud = proxy.hpcell
13   hadoop isa HadoopMaster { }
14 }
15 slave1 isa VM {
16   in_cloud = proxy.hpcell
17   hadoop isa HadoopSlave { }
18     master = master.hadoop
19   }
20 }
21 slave2 extends slave1
```

[13] We cannot provide an example of this module due to copyright issue.

```
22  slave3 extends slave1
```

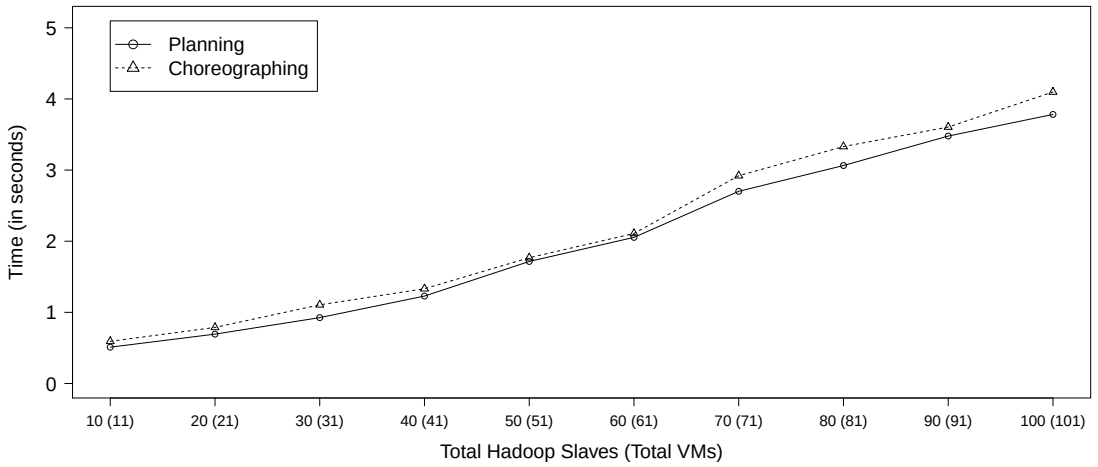


Figure 6.34: Planning: time of Nuri for generating a partial-order plan in orchestration mode. Choreography: time of Nuri for generating BSig models in choreography mode.

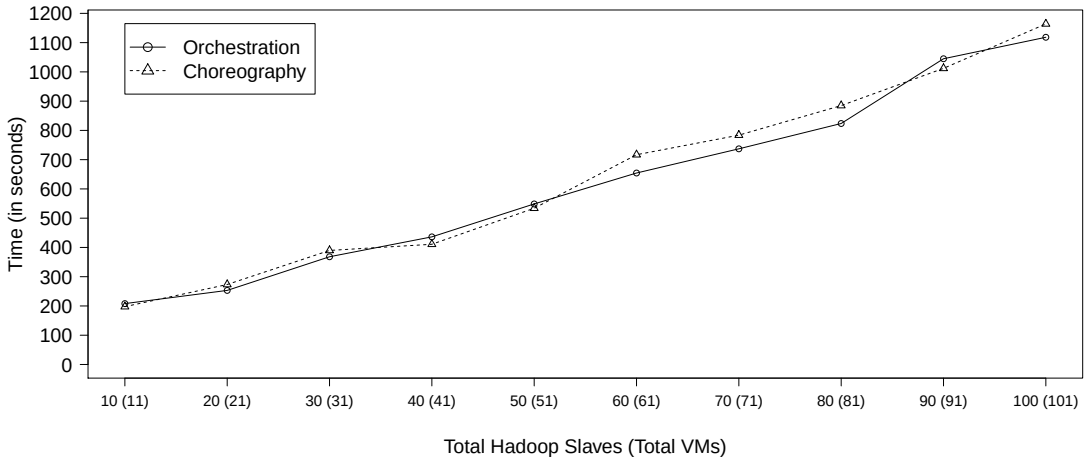


Figure 6.35: Execution time of Nuri for deploying Apache Hadoop system from scratch on HP Cells.

To measure the performance of Nuri, we created 10 different configurations of a Hadoop system that has one master and different number of slaves: 10 slaves, 20 slaves, ..., 100 slaves – the master and the slaves were installed and run on different virtual machines. For each configuration, we ran 10 experiments two times, once with orchestration and another in choreography mode, and then we calculated the average time for orchestration and choreography modes. We used a machine with the specifications Intel CPU 1 core, 2.4 GHz, 8 GB of memory, and Linux operating system as the Nuri master. Each VM will have 1 CPU core, 4 GB of memory, and Linux operating

system. We activated the multi-heuristics (see §4.2.3) of the Nuri planner. Another machine was set as the cloud proxy that controls an HP Cells account – the machine has a Nuri agent with the cells module.

Figure 6.34 show the planning and the choreographing average times from 10 experiments. The x-axis shows the number of Hadoop slaves where the numbers in bracket are the number of virtual machines, while the y-axis shows the planning and the choreographing times in seconds. Notice that the choreographing times are slightly higher than the planning time, which is an obvious result since the choreographing process consists of a planning and a BSig model generation. The planning times themselves are likely to be linear to the number of slaves. For the largest case where the system has 1 master and 100 slaves, the partial-order workflows with 404 actions were generated in average 3.78 seconds, while the BSig models were generated in average 4.10 seconds.

Figure 6.35 show the average deployment times from 10 experiments using orchestration and choreography techniques. Notice that both deployment times are similar and likely to be linear to the number of slaves. For the largest case, the orchestrations were finished on average 1118.28 seconds, while the choreography were finished on average 1163.92 seconds. These show that there is no significant difference between the orchestration and the choreography techniques. We suspect that this is due to very low network latency time ($<$ 0.1s) between the master with the managed VMs. We believe that the result could be different if the network latency time is higher.

However, if there is a problem on Nuri master such as network outage, then the workflow execution with orchestration will be stopped, and the system will be out of control. On the other hand, the execution with choreography will not stop because the agent does not depend on the master. This is clearly increasing the robustness of the system.

We found that there were several bottleneck issues that affected the deployment times. First, the backend storage of HP Cells were creating the VMs likely in linear time[14], in the sense that only n VMs can be created at the same time. Thus, although the workflow has partial ordering constraints (e.g. actions for creating the VMs were mutually exclusive), and we allowed the Cell module to send parallel requests to the HP Cells service for creating 100 VMs, but these requests were completed in linear time by the HP Cells. Second, we used a single repository of Hadoop software package which should be downloaded by the Apache module during installation, where the size of the Hadoop software package is 180 megabytes. Thus, the download time were

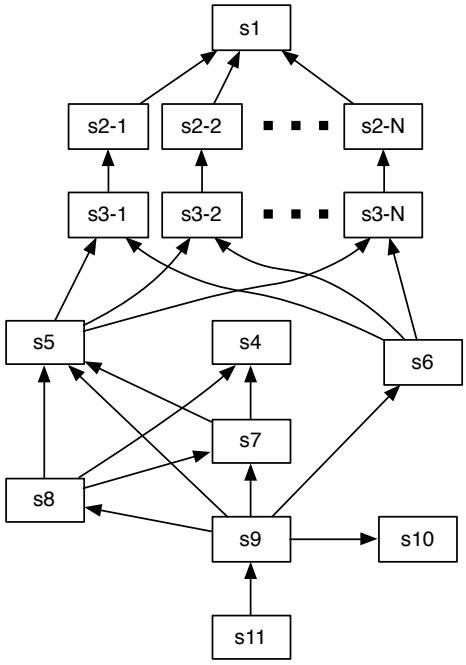


Figure 6.36: The architecture of the HP IDOLoop system, where the arrows show the dependencies between the services.

decreasing when the number of download requests increased because the maximum throughput (including storage and network latency) of the repository was 12.5 MB/s.

During execution, the ordering constraints of the workflows were maintained by Nuri (either orchestration or choreography) – for example: the Hadoop slaves will not be started until the master has been started. We verified this by checking the timestamp of the action executions in the log files (when the executable was started and finished), and then compared them across different machines. In addition, we run several Hadoop applications on the system to ensure that the system was working correctly – for example: we run *wordcount* application available in Hadoop software distribution.

6.4.3 Use Case 2: HP IDOLoop

The HP IDOLoop system [Hewlett-Packard, 2014] is a proprietary software of Hewlett-Packard for processing large numbers of documents to extract useful information such as texts, faces, and barcodes. It consists of a set of inter-connected services that form

[14]This linear time is not caused by Nuri implementation since the requests of VM creation were sent in parallel to HP Cells. We suspect that the installation of HP Cells used in this experiment had *n* number of threads that serve volume creation requests which are required by the VMs, where each thread can only create one volume at a time.

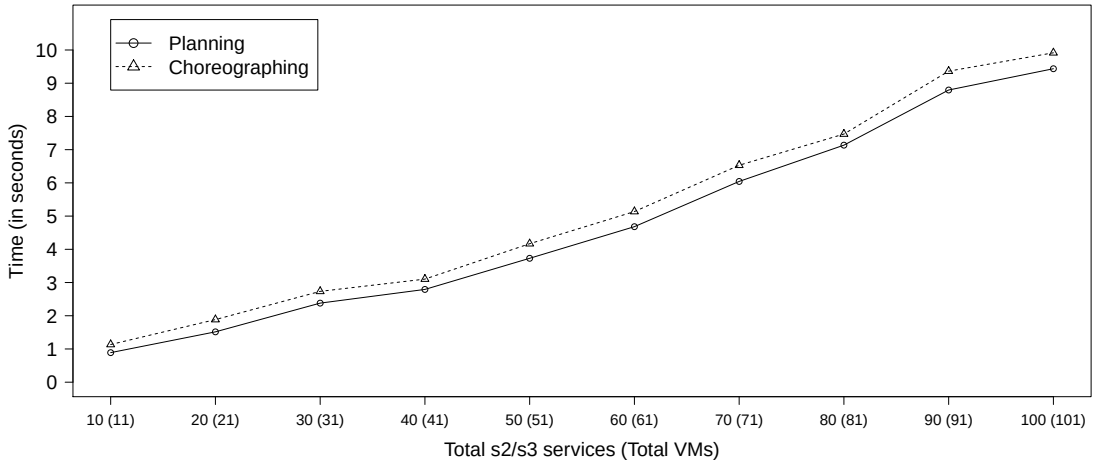


Figure 6.37: The Nuri planning time for deploying HP IDOLoop system from scratch on the HP Cells.

streams of processes. Figure 6.36 illustrates the architecture of the system, which looks like a combination of system-B and C. The nodes in the figure are services, which are running on virtual machines, and the arrows are the dependencies between services. The system can be scaled-up or scaled-down by increasing or decreasing the number of services of s_2/s_3. Due to the proprietary issue, we cannot describe more details about this system.

Since there are 11 types of services $(s_1, ..., s_{11})$, then we created 12 Nuri modules that have capabilities to install/uninstall, configure and start/stop all type of IDOLoop services on particular machines. One module contains common configurations and actions shared between all services through inheritance, while the others are specific module that contains specific configurations and actions for particular service.

In this experiment, the HP IDOLoop system was deployed from scratch to the HP Cells cloud infrastructure. We created 20 different configurations of HP IDOLoop system that have different number of s_2/s_3: 10, 20 slaves, ..., 100 – service s_2 and s_3 must be on the same machine while other services were on different machines.

A machine with the specifications of Intel CPU 1 core, 2.4 GHz, 8 GB of memory, and Linux operating system was used for the Nuri master. The multi-heuristics (h^{FF} and h^{LM}) and the partial-order plan generated of the Nuri planner were activated. Another machine was set as the cloud proxy that controls an HP Cells account. Unfortunately, we cannot present the example of specification in this thesis due to the proprietary issue.

Figure 6.37 show the planning and the choreographing average times from 10 ex-

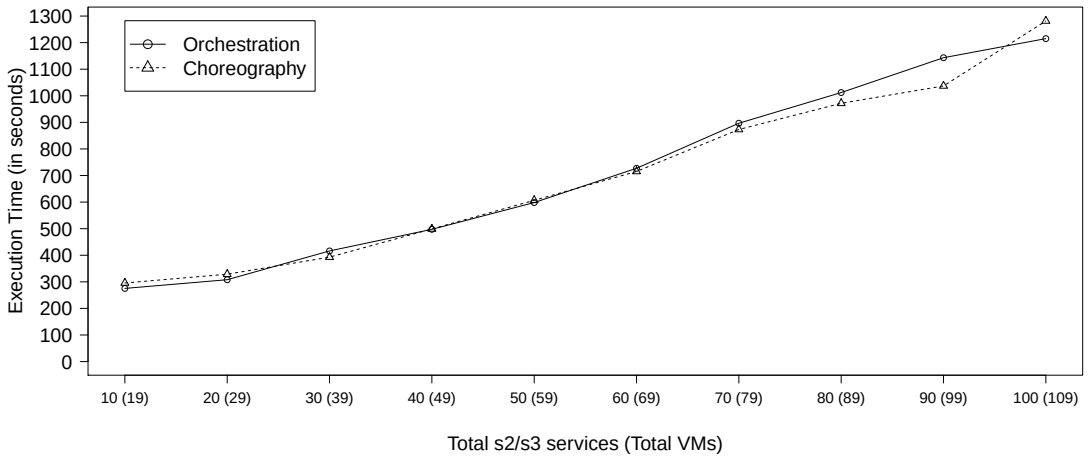


Figure 6.38: The Nuri execution time for deploying HP IDOLoop system from scratch on the HP Cells.

periments. The x-axis show the number of s_2/s_3 services where the numbers in bracket are the number of virtual machines, while the y-axis show the planning and the choreographing times in seconds. Notice that the choreographing times are slightly higher than the planning time, which is similar to the Hadoop experiment results. The planning times are linear to the number of slaves. For the largest case where the system has 109 VMs, the Nuri planner can generate the partial-order workflows with 456 steps in average 9.43 seconds, while the BSig models were generated in average 9.92 seconds.

Figure 6.38 show the average deployment times from 10 experiments using orchestration and choreography techniques for deploying the IDOLoop system with various configurations. Notice that both techniques have similar deployment times. For the largest case, the orchestrations were finished in average 1215.15 seconds, while the choreography were finished in average 1281.49 seconds. Similar to the Hadoop results, there is no significant difference between the execution times of both techniques.

The experiments also have similar bottleneck issues as found in the Hadoop experiments. During execution, the dependencies between services were maintained by Nuri (either orchestration or choreography) – for example: service s_9 was started after starting service s_5, s_6, s_7, s_8, and s_{10}. We verified this by checking the timestamp of action executions in the log files (when the execution was started and finished), and then compared them across different machines.

6.4.4 Use Case 3: Configuration Relocation on the BonFIRE Infrastructure

In this experiment, we performed the configuration *relocation* process between cloud infrastructures. The term *relocation* refers to a process that moves services between different cloud infrastructures, reliably and without disrupting the operation of the overall service during the transfer process [Herry and Anderson, 2013]. This is a complex operations; live VM migration is not normally possible between different sites that use different platforms, or they are controlled by different organisations, so new copies of the services must be instantiated on the new infrastructure, and any clients of the service must be reconfigured to reference the new service, before we can tear down the old service. This is likely to involve some reconfiguration of the services themselves (as a minimum, the IP addresses will change). Any significant distributed application will require a more complex reconfiguration, and a carefully crafted workflow to preserve a running service during all stages of the transfer.

The experiment was using the BonFIRE infrastructure [Kavoussanakis et al., 2013]. It provides an ideal testbed on which to explore the relocation process; it is a federated cloud infrastructure with heterogeneous platforms – for example, HPLabs[15], Inria, and EPCC[16] sites, are all maintained independently by different teams, using different technologies (Cells and OpenNebula). There is no capability for transferring virtual machines between sites – even though EPCC and Inria sites are using the same platform i.e. OpenNebula, but a virtual machine cannot be transferred between them. However, there is a common broker that can be used to manage virtual machines on multiple sites. This broker

We evaluated a scenario where there are two 3-tier systems; the first system has version number 1, running on the EPCC BonFIRE site, and being used by a number of clients; the second system has version number 2, is running on the Inria BonFIRE site, and being tested by the engineers. Since the second system has passed all tests, then we would like to relocate this version to the EPCC site to replace the older system. Figure 6.39 illustrates these current and desired states.

Although the software is deployed independently on different VMs, the relocation process must satisfy some global constraints due to service dependencies. In addition, we do not want to disturb any usage of the service by the clients during the process.

[15]Hewlett-Packard Laboratory
[16]The Edinburgh Parallel Computing Center

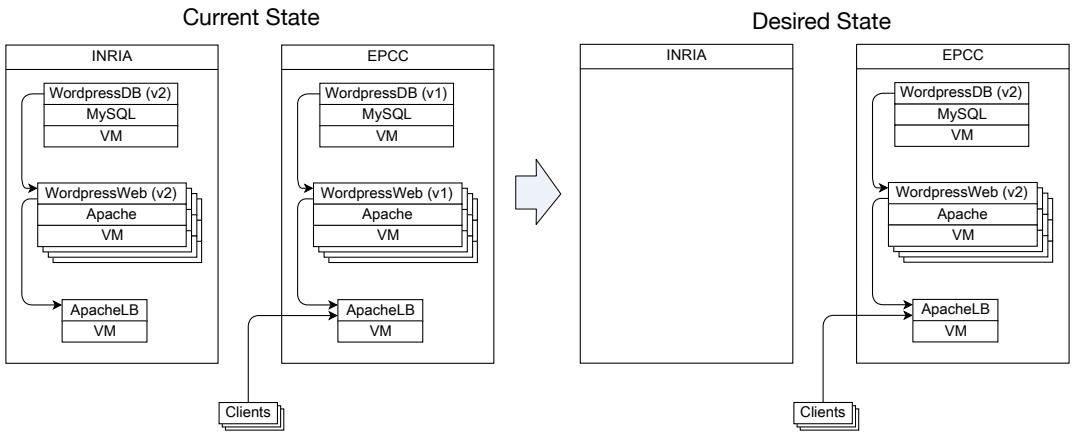


Figure 6.39: The current state (left), and the desired state (right) of the 3-tier system.

Thus, the following global constraint should be maintained:

- The web services depend on the database service: whenever the web services are running then the database service must be running as well;
- The load balancer depends on the web services: whenever the load balancer is running then the web service must be running as well;
- All clients should always refer to a running server.

```
1  vm_epcc isa VM { created = true; in_cloud = proxy.epcc; }
2  vm_inria isa VM { created = false; }
3  main {
4    proxy isa Machine {
5      sfpAddress = "172.18.240.38"
6      // proxy component for EPCC site
7      epcc isa Bonfire  { location = "uk-epcc"
8                          experiment = "autocloud"; }
9      // proxy component for INRIA site
10     inria isa Bonfire { location = "fr-inria"
11                         experiment = "autocloud"; }
12   }
13   pc isa Client {
14     sfpAddress = "172.18.240.39"
15     refer = vm21.apache // change reference to the latest system
16   }
17   // "virtually" move machines of the latest system to EPCC
18   // site by setting "in_cloud" with value "proxy.epcc"
19   vm21 extends vm_epcc {
20     apache isa Apache {
21       running = true
22       is_load_balancer = true
23       lb_members = [vm22.apache]
24     }
25   }
```

```
26    vm22 extends vm_epcc {
27      apache isa Apache { running = true; }
28      wp_web isa WordpressWeb {
29        version = 2
30        installed = true
31        http = vm12.apache
32        database = vm23.wp_db
33      }
34    }
35    vm23 extends vm_epcc {
36      mysql isa Mysql { running = true }
37      wp_db isa WordpressDB {
38        version = 2
39        installed = true
40        mysql = vm23.mysql
41      }
42    }
43    // delete machines of old system
44    vm11 extends vm_inria {
45      apache isa Apache { }
46    }
47    vm12 extends vm_inria {
48      apache isa Apache { }
49      wp_web isa WordpressWeb { }
50    }
51    vm13 extends vm_inria {
52      mysql isa Mysql { }
53      wp_db isa WordpressDB { }
54    }
55    // global constraints
56    global {
57      // pc always refers to a running system
58      pc.refer.running = true
59      // dependencies between services
60      if vm11.apache.running = true then vm12.apache.running = true
61      if vm12.apache.running = true then vm13.mysql.running = true
62      if vm21.apache.running = true then vm22.apache.running = true
63      if vm22.apache.running = true then vm23.mysql.running = true
64    }
65  }
```

To implement this relocation, we defined the desired state and the global constraints in SFP. The above specification is an example of the desired state with one application service. By submitting this specification to the Nuri master, a workflow is generated as shown in figure 6.41. The execution of this plan relocated the new version system from Inria to EPCC, configured the client to use this system, and finally deleted the old version system from EPCC site. The plan execution did not violate any global constraints – the services were started in the correct order, and the client was redirected to use another running service before the old one was deleted.

In the experiments, we used a virtual machine with 2 CPUs and 2 GB RAM in

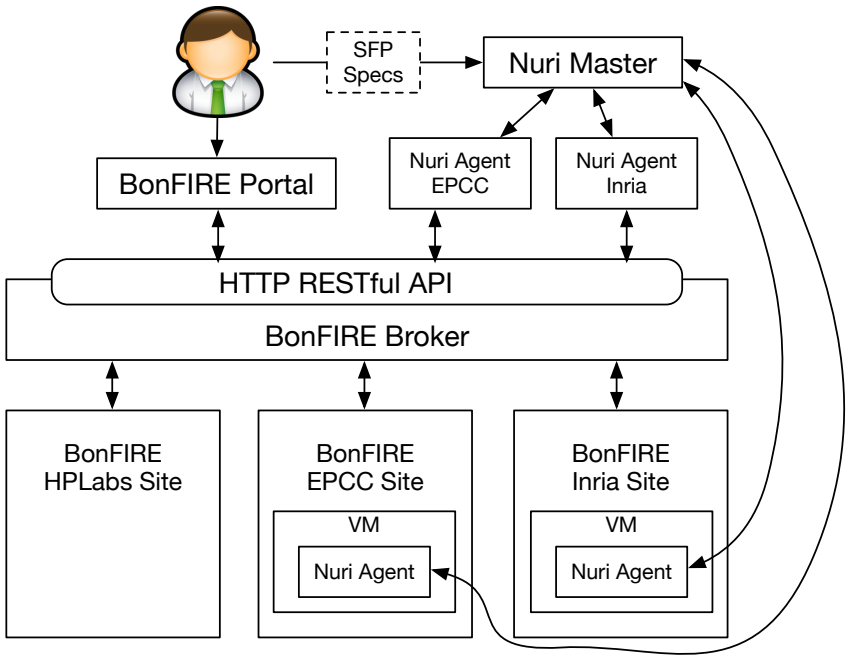


Figure 6.40: Nuri on BonFIRE infrastructure.

EPCC site as the Nuri master. Each BonFIRE site was managed by an instance of proxy module. This proxy module uses a *Restfully* Ruby library [bon, 2014] to communicate to the BonFIRE broker for getting VM's status, creating or deleting VM on a particular site. There are two nuri agents, one has an instance of proxy module controlling Inria site, while another has an instance controlling EPCC site. For the managed system, we used *small* (1 CPU, 1 GB RAM) instance VMs. For the software stack, we used unmodified Debian Squeeze 10G v5, Apache Web Server, MySQL Database Server, and the Wordpress Content Management System (CMS). Every software is controlled by a Nuri module, except Wordpress CMS which is controlled by two modules: 1) WordpressWeb that controls the *application* layer, 2) WordpressDB that controls the *database* layer. Separating the application and database layers of Wordpress CMS into two modules allows us to increase (or decrease) the number of application services which the system can have. There is another Nuri module which manages the client configuration. This module is capable of redirecting service references in a similar way to a DNS lookup[17]that is by modifying the IP address registered in a particular file. All VMs involved in the experiment were connected to the BonFIRE WAN network.

Figure 6.40 illustrates the interactions between Nuri agents and BonFIRE broker

[17]Commonly, a Linux operating system has file `/etc/resolve.conf` that contains a list of IP addresses of DNS servers which can be used to resolve a host name.

and infrastructures. For example, for creating a VM, the Nuri agent will find the implementation of action `create_vm` in the proxy module. Invoking this action would send a request to the BonFIRE broker through *Restfully* library. This library encapsulates the request into an HTTP POST request where the specification of the VM (e.g. the number of CPU) is serialized into JSON data format and sent to the broker web service. The agent will then wait until the broker sends a response that the VM is ready. Whenever such response is received, the agent asks the broker to send the VM's configuration data such as its IP address. This IP address is used by the agent to install a software package of Nuri agent into the new VM using remote shell execution and start the Nuri agent daemon – note that the experiment used an unmodified VM image which does not have an installed Nuri agent software package. Finally, the agent sends a broadcast message containing the new VM's name and IP address to other agents to notify them that a new agent has been running. This allows other agents to update their agent-database and perform a peer-to-peer communication with the new agent.

We ran several experiments using the same scenario but with different number of services of application layer (WordpressWeb). Each experiment was run five times and then took the average time. Figure 6.42a illustrates the comparison of the planning and the choreographing times between various numbers of services. Notice that the planning and the choreographing times are linear to the number of application services, where the latter is slightly higher than the former due to an additional process to translate the workflows to the BSig models. For the largest case where each system has 5 application services, the partial-order workflows were generated in average 6.37 seconds, while the BSig model were generated in average 7.03 seconds.

Figure 6.42b shows the comparison of the deployment times between the orchestration and the choreography techniques. For the largest case, the desired state were achieved in average 1487.56 and 1539.30 seconds using the orchestration and the choreography respectively. This shows that there is no significant difference on the deployment times of both techniques. However, the orchestration requires the master to be on our private infrastructure, or on any cloud, and it must be always connected to the VMs. Obviously, if there is a network outage on the master's infrastructure, then the whole execution will be stopped. But in choreography, the execution of the system on a healthy cloud infrastructure will continue even if there is a problem on the master's infrastructure.

The orchestration execution time is near optimal since the Nuri orchestrator implements a partial-order execution algorithm – it allows mutually exclusive actions to

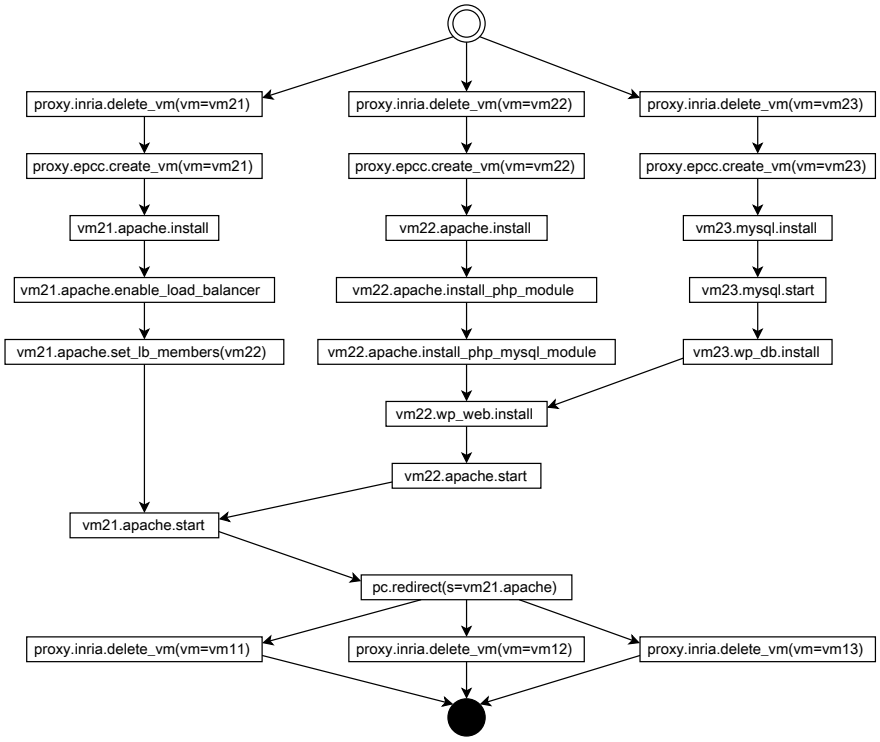


Figure 6.41: The generated workflow for the system with one application service.

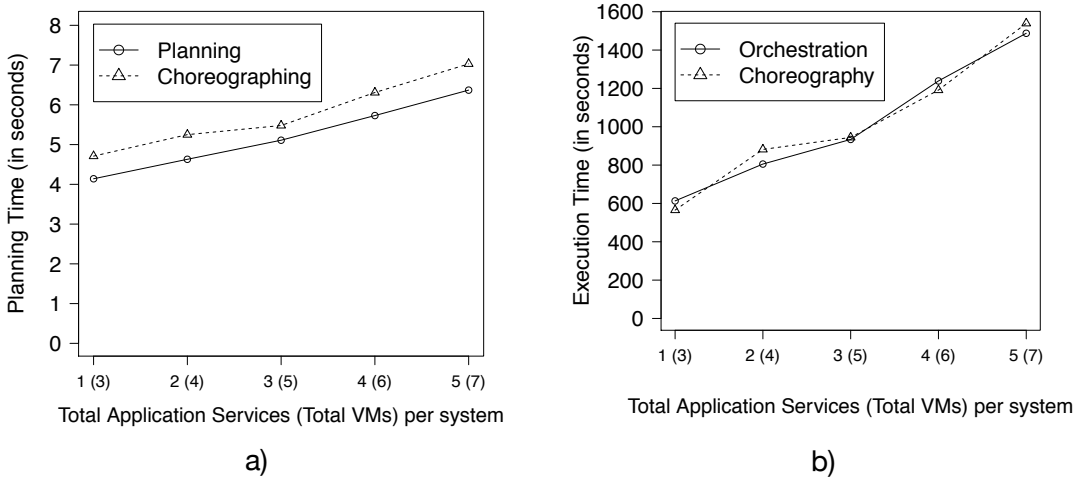


Figure 6.42: The planning (a) and the execution (b) times with various number of application services (per system).

be executed in parallel. This enabled the orchestrator to send the execution requests of mutually exclusive actions to agents simultaneously, for example: installing a software package on different machines. Hence, the target agents can execute the actions in parallel. A similar thing was achieved when we used the choreography technique. Since every agent has its own goals and actions, then the agents can then select their appropriate action simultaneously. If the actions are mutually exclusive, these agents

will execute the actions in parallel. However, figure 6.42 shows an unexpected result: the execution time is linear with the number of VMs. We received a confirmation from the EPCC site owner that there is an issue on OpenNebula platform used by EPCC and Inria sites i.e. the storage manager could only create two VM images in parallel.

Network latency between Nuri master and agents can affect the deployment time. However, during experiments, we found that the BonFIRE WAN network is very stable and fast[18]. In addition, the sizes of messages which are sent from Nuri master to agents or between the agents are small – the size of message that contains an action description sent by Nuri master is less than 500 bytes, while the size of message than contains remote preconditions sent by one to another agent is less than 200 bytes. Each VM has a network interface whose throughput is 12.5 MB/seconds. Thus, the overall overhead communication is quite low.

Self-Healing

We also tested the self-healing capability of the agent when the choreography mode was activated. We manually stopped or uninstalled some services on a running system randomly and checked the state of the system several minutes later. Since every agent continuously executed the cooperative reactive regressions (CRR) algorithm, then it checked the state of the VM periodically and detected such errors as goal flaws, selected and invoked some local actions (it may request remote preconditions to other agents) to fix the flaws.

In another, we manually deleted some random VMs on the EPCC site. In this case, since the cloud proxy of EPCC site was continously executing its local model, it detected that those VMs did not exist. Based on the model, it then automatically created some VMs to replace the deleted ones. After installing and starting Nuri agent on each new VM, the cloud proxy agent sent the appropriate local model to be executed on the new VM. After several minutes, all deleted VMs had been replaced with the new ones and the system was again stable. These results show that the choreography technique enables the agents to automatically fix a drift from the desired state by using the existing BSig model, distributively, and without any *re-choreographing* process.

[18]It required less than 50ms to ping from one to another VM either in the same or different sites.

6.4.5 Discussion

The current implementation of Nuri orchestrator is using a progressive execution algorithm. On the other hand, in *choreography* mode, the agents are using the regression execution algorithm (see §5.2.3) for executing a workflow. The above experiment results indicate that there is no significant difference on the execution times between these algorithms.

On the other side, *orchestration* and *choreography* use different mechanisms to enable self-healing capability. Due to the progressive execution, when Nuri is in *orchestration* mode, it has to perform a re-planning in order to generate a new workflow to fix any drift from the desired state. This re-planning is required because the previously generated workflow is not applicable when using progressive execution algorithm – the current state of the system is not the same with the initial state of the workflow. On the other hand, whenever Nuri is in *choreography* mode, the agents can try to re-use the existing actions of BSig model using regression execution to fix the drift. If they fail, then Nuri performs re-choreographing[19]. Thus, *choreography* mode can help Nuri to reduce the necessity to perform re-planning. Of course, it is possible to implement the regression execution algorithm for single agent (see figure 5.7) into Nuri orchestrator. This allows the *orchestration* mode of Nuri to re-use the actions of previously generated workflow in order to fix the drift.

If we are comparing the above execution times, we might not see any benefit on using *choreography* comparing to *orchestration*. This is mainly because the planning time is not significant – for example, workflows of the above experiments can be generated in less than 10 seconds (see figure 6.34, 6.37 and 6.41). Hence, re-planning only contributes a very small percentage of time required to "heal" the system which involves both planning and plan execution. We suspect that *choreography* will perform better than *orchestration* for a larger or more complex system.

On the other hand, *choreography* offers a better resilient comparing to *orchestrator*. This is because whenever we use *orchestration* and there is a problem on Nuri master, such as network outage, then any drift on the managed system cannot be fixed immediately – it has to wait until the problem has been resolved. This is not the case for *choreography* since the agents can automatically fix the drift even though there is a problem on Nuri master.

[19] A re-choreographing process requires re-planning.

6.4.6 Summary

To summarise, the above experiments have shown that:

- The Nuri configuration tool that implements the technique described in §4.2 can be used in practice to generate workflows for configuring real systems;
- The choreographing times are slightly higher than the planning times due to an extra step for translating the workflow to the BSig models;
- The deployment times using orchestration are near optimal since the Nuri generated partial-order workflows, and then executed them with a partial-order execution algorithm;
- There is no significant difference between the orchestration and the choreography techniques. However, the orchestration execution will stop if there is a failure on the Nuri master. But in choreography, the execution of the system on a healthy infrastructure will continue even if there is a problem on the master's infrastructure.
- The choreography technique enables the agents to automatically fix a drift from the desired state by using the existing BSig model, distributively, and without any *re-choreographing* process.

Chapter 7

Conclusion

Declarative specification approach has been widely accepted as the most appropriate one for managing configurations of cloud systems – it promises that the user can explicitly specify the desired state of the system, and the tool will automatically compute and execute the necessary actions to bring the system from current to this desired state. However, this promise is yet fully accomplished by state-of-the-art practical declarative tools because they cannot maintain necessary (ordering) constraints during configuration changes. If they can, then the user must manually define the ordering constraints in the specification, which is conceptually similar to imperative scripts.

This thesis has shown that the automated planner is the missing link of practical declarative tools in order to complete the promise. By making use of the planner, the tool can automatically generate a workflow between any two viable states, enabling unattended reconfiguration. The workflows are guaranteed to achieve the desired state while preserving the necessary constraints. With the choreography technique, the tool can construct a set of reactive agents which executes the workflow without any central controller. This enables the agents to form a *self-healing* system which increases its resilience while retaining a predictable and deadlock-free workflow. The development and the experiment results of Nuri system configuration tool, which implements this approach, have proved that this is practically visible and scalable.

In addition, this thesis has demonstrated that the formalisation of a practical configuration language is critical for finding and correcting problems with the existing compiler, and proving necessary properties of the language. It also provides a consistent, platform-independent reference for extending the language as well as developing alternative implementations.

7.1 Hypotheses and Contributions Revisited

Along the journey for completing the promise of declarative specification approach, this thesis has achieved several milestones. First, it has developed the formal semantics of the SmartFrog configuration language. This development has considerably increased our understanding of the language, highlighted difficult areas, and identified problems with a production compiler. It also has been a valuable guide to the development of the language extensions and the corresponding practical compiler. This formal semantics confirms our first contribution which is mentioned in the introduction.

In the second milestone, this thesis has been able to extend the core syntax of the SmartFrog language to adding new features that allow the administrator to declaratively define a configuration specification which consists of the desired state, the actions, and the global constraints. In addition, a static-type system has been developed to provide a type level safety which can be checked before deployment. This increases our confidence that the specification is correct, and also helps the planner to reduce the search space. This second milestone confirms that **hypothesis 1** of this thesis has been achieved, and also supports our second contribution.

The third milestone is the development of a domain independent technique that compiles a planning problem with extended goals into a classical planning problem. This technique has been tested using planning problems from International Planning Competition. The experiment results show that the planner which implements this technique has better planner times and coverage comparing to the MIPS-XXL planner. This confirms the third contribution of this thesis.

In the forth milestone, this thesis has developed a technique to translate a configuration task into a classical planning problem, which can be solved by a classical planner. The evaluation results of Nuri configuration tools, that implements this technique, have clearly shown the advantages of automated planning for cloud systems reconfiguration – the workflows can be automatically generated between any two declarative states within a reasonable time, enabling unattended and autonomic reconfiguration for failure recovery or other reasons. The generated workflows are guaranteed to achieve the desired state, while preserving the necessary (ordering) constraints at every state of reconfiguration process. This confirms **hypothesis** 2, and our forth contribution of this thesis.

In the last milestone, this thesis has introduced an alternative technique for deploying the configuration, which is called as *choreography*. This technique translates a

workflow into a set of Behavioural Signature models, each of which is sent to a particular agent. The evaluations have shown that the agents, which executes the models using the Cooperative Reactive Regression (CRR) algorithm, can autonomously execute the workflow in a distributed way, while preserving the necessary (ordering) constraints. This can eliminate a single point of failure, and hence increases the system's resilience. The evaluations have also shown that the agents can fix particular drifts from the desired state without the need of re-planning. These results confirm **hypothesis** 3 of this thesis, and also our fifth and sixth contributions.

7.2 Future Works

Several parts of this thesis can lead us to some potential future works. First, although the static-type system of SFP language has been presented, but the type checking functions for the global constraints and actions have yet been defined. The absence of these functions makes the current compiler unable to detect type-errors in constraints or actions' preconditions/effects. A common treatment whenever there is a type-error is that the compiler assumes that the whole specification is incorrect, and then immediately stops the compilation process. However, we could have a different behaviour for handling this kind of errors. For actions, it is possible that the compiler automatically removes all actions with type-error from the specification. For constraints, the compiler can assume that the valuation value of a constraint with type-error is false. With these alternative treatments, the compiler can still produce the final output of the specification, but with warnings. A further study on these different semantics is required to determine which one is better in practice.

Previous section has mentioned advantages on formalising the semantics of a practical configuration language. We believe that similar advantages can be also obtained by applying the same process (see figure 3.3) on the others e.g. Puppet language [Puppet Labs, 2014]. The impact of this work can be very significant, in particular if the language is used to define specifications of critical systems. Unfortunately, most of practical configuration languages are lack of good documentations. The author of the language may not want to share critical information due to some restrictions e.g. copyright. Another challenge is that the language itself is *evolving* from one to another version during the formalisation process, where some new features are added and some others are deprecated. Thus, it is essential for us to find a *stable* core features of the language as the starting point of this process. Afterwards, we can expand the semantics

by adding other features one-by-one.

Basically, Nuri is using a classical planner as its low-level solver. Since this classical planner is using FDR, then all planning problems must be converted to it. FDR has help researcher in developing powerful heuristic techniques which significantly increase the performance of the planners. Unfortunately, it only allows the goal or the preconditions of actions to be expressed as a conjunction of atoms. This restriction implies that any complex formula of goal or preconditions must be converted into an equivalent Disjunctive Normal Form (DNF) formula. Unfortunately, this conversion can produce an exponential size of DNF formula comparing to the original one. One of possible solution to this issue is that we can allow a complex formula to be used in the goal or the preconditions. This avoids the explosion of the size of the DNF formula since the conversion is not required. However, there is a new challenge here: a new heuristic technique must be developed – indeed existing heuristics cannot be used due to different representations. But, it is possible to adapt the principles of the heuristics to this new representation.

Although the regression execution can increase the viability of a plan, but it is possible that the current of the system is not equal to the initial or intermediate states of the plan. This will invalidate the existing Behavioural Signature model which implies a re-planning process. One of approaches for this problem would be increasing the validity of the model by embedding multiple plans into a single model. Although this may likely arise a livelock or deadlock situation in the model, but incremental merging process with automated livelock/deadlock detection could be used to avoid this situation. Another approach is reformulating the problem as a non-deterministic planning problem, and then using a *non-classical* planner to generate a conditional plan e.g. [Bertoli et al., 2001, Hoffmann and Brafman, 2005]. More recently, [Albore et al., 2009, Bonet and Geffner, 2011, Brafman and Shani, 2012] describe very interesting techniques – they exploit classical planners to generate such conditional plan by translating the problem into a classical one under several restrictions. It would be interesting to get the comparison results of these two approaches.

There is a case where a centralised planning approach, which is used in this thesis, cannot be applied. One of them is when the system is consisting two subsystems, each of which is controlled by two different organisations who do not trust each other, and there is no such a trusty third party. Solving a planning problem in this kind of situations requires a new approach i.e. *multiagent planning*. There are various types of multiagent planning problems. Perhaps the most relevant with reconfiguration problem

of cloud systems would be *Shared-Goal Multiagent Planning* – there is a single goal where the objective is to find a plan that achieves the goal without taking into account actions performed by each agent [Crosby, 2014]. Some early results show that the heuristic used to solve this problem has a promising performance. Further investigation is needed to evaluate this approach using practical reconfiguration problems.

Appendix A

SmartFrog Language

A.1 Concrete Syntax

SF	::=	B
B	::=	A B | ε
A	::=	R V
P	::=	R | `{` B `}`
PS	::=	P (`,` P)* | ε
V	::=	BV `;` | LR `;` | `extends` PS
R	::=	I (`:` I)*
DR	::=	`DATA` R
LR	::=	R
Vec	::=	`[` (BV (`,` BV)* | ε) `]`
Null	::=	`NULL`
Bool	::=	`true` | `false`
BV	::=	Bool | Num | Str | DR | Vec | Null

A.2 Proofs

Proposition 3.13. Assume $s \in \mathcal{S}$ has unique identifiers i.e. $\forall \langle id_i, v_i \rangle, \langle id_j, v_j \rangle \in s \; . \; i \neq j \Rightarrow id_i \neq id_j$. Then operation $s' = put(s, id, v)$ always returns $s' \in \mathcal{S}$ that also has unique identifiers i.e. $\forall \langle id'_i, v'_i \rangle, \langle id'_j, v'_j \rangle \in s' \; . \; i \neq j \Rightarrow id'_i \neq id'_j$.

Proof. The proof is given by induction.

Basis: Show that the statement holds for $s = \varnothing_{\mathcal{S}}$.

Since $s = \varnothing_{\mathcal{S}}$, then the first equation in definition 3.12 is applied.

$$\begin{aligned} s' &= put(\varnothing_{\mathcal{S}}, id, v) \\ &= \langle id, v\rangle :: \varnothing_{\mathcal{S}} \end{aligned}$$

Since there is only one pair in s', then it is valid to say that s' has a unique identifier. Thus, it has been shown that the statement holds for $s = \varnothing_{\mathcal{S}}$.

Inductive step: Show that the statement holds for any arbitrary $s \in \mathcal{S}$ where s has unique identifiers. There are two cases:

- First is when $s = \langle id, v_s\rangle :: s_p$. Since this will match with the second equation of definition 3.12, then it must be shown that $s' = \langle id, v\rangle :: s_p$ has unique identifiers. The equation has checked that the identifier of the first element of s is equal with id. Since s has unique identifiers, then id will not in s_p. This, it is valid to state that $s' = \langle id, v\rangle :: s_p$ has unique identifiers.

- Second is when $s = \langle id_s, v_s\rangle :: sp$. Since this will match with the third equation of definition 3.12, then it must be shown that $s' = \langle id_s, v_s\rangle :: put(s_p, id, v)$ has unique identifiers. Since s has unique identifiers then s_p also has unique identifiers. Because the statement holds, then operator $put(s_p, id, v)$ will return a store has unique identifiers. id_s itself will never exist in the store returned by $put(s_p, id, v)$ because id_s is not exist in s_p (if it doest exist, then the second equation will be called, not the third). This, it is valid to state that $s' = \langle id_s, v_s\rangle :: put(s_p, id, v)$ has unique identifiers.

Since the basis and the inductive step have been performed, then by mathematical induction, the statement in proposition 3.13 holds. □

Proposition 3.16. Assume $s \in \mathcal{S}$ then $\forall r \in \mathcal{R}\,.\, find(s, r) \neq \perp \Rightarrow find(s, prefix(r)) \in \mathcal{S}$.

Proof. There are three cases that needs to be considered. The first case is when $r = \varnothing_{\mathbb{I}}$ where $prefix(r) = \varnothing_{\mathbb{I}}$. Based on the first equation of definition 3.15, since $find(s, \varnothing_{\mathbb{I}}) = \varnothing_{\mathcal{S}}$ then $find(s, prefix(r)) = \varnothing_{\mathcal{S}}$. Because $\varnothing_{\mathcal{S}} \in \mathcal{S}$, then it is valid to say that the statement holds for $r = \varnothing_{\mathbb{I}}$.

The second case is when $r = id :: \varnothing_{\mathbb{I}}$ where $prefix(r) = \varnothing_{\mathbb{I}}$. It is similar with the first case where $find(s, prefix(r)) = \varnothing_{\mathcal{S}}$ and $\varnothing_{\mathcal{S}} \in \mathcal{S}$. Thus it is valid to say that the statement holds for $r = id :: \varnothing_{\mathbb{I}}$.

The last case is when $r = id_1 :: ... :: id_{n-1} :: id_n :: \varnothing_{\mathbb{I}}, n \geq 2$ where $prefix(r) = id_1 :: ... :: id_{n-1} :: \varnothing_{\mathbb{I}}$. Without loosing any details, we can rewrite $find(s, r) = find(find(id_1 :: ... :: id_{n-1} :: \varnothing_{\mathbb{I}}), id_n :: \varnothing_{\mathbb{I}})$. Since $find(s, r) \neq \perp$ then the first branch's condition of

the last equation in definition 3.15 should be satisfied i.e. $s.1 = id \wedge s.2 \in \mathcal{S}$, where $find(id_1 :: ... :: id_{n-1} :: \varnothing_{\mathbb{I}}) = s.2$. Because the condition ensures that $s.2 \in \mathcal{S}$, then it is valid to say that the statement holds for $r = id_1 :: ... :: id_{n-1} :: id_n :: \varnothing_{\mathbb{I}}, n \geq 2$.

Since the statement holds for every case then the statement in proposition 3.16 holds. □

Proposition 3.18. Assume $s \in \mathcal{S}$ where s has unique identifiers, and $\forall s_i \subset_{\mathcal{S}} s : s_i$ has unique identifiers. Then operation $s' = bind(s, id :: r, v)$ always returns s' that has unique identifiers and $\forall s_j \subset_{\mathcal{S}} s' : s_j$ has unique identifiers as well.

Proof. The proof is given by induction.
Basis: Show that the statement holds for any arbitrary store s and value v, while the target reference $id :: \varnothing_{\mathbb{I}}$.
For this case, the second equation of definition 3.14 is applied, where: $s' = put(s, id, v)$. Since the premise specifies that s has unique identifiers, then based on proposition 3.13 $put(s, id, v)$ always returns a store that has unique identifiers for any arbitrary id and v. Thus, it has been shown that the statement holds for any arbitrary s and v, and the target reference $r = id :: \varnothing_{\mathbb{I}}$.

Inductive step: Show that the statement holds for any arbitrary store s, value v, and reference $id :: r$. There are two cases:

- First is when $s = \langle id, v_s \rangle :: s_p$. Since this will match the forth equation of definition 3.14, then there are two branches:
  - When $v_s \in \mathcal{S}$, it must be then shown that $s' = \langle id, bind(v_s, r, v) \rangle :: s_p$ has unique identifiers. Since the premise states that s has unique identifiers, then no element in s_p whose identifier is equal to id. Based on the premise as well, v_s has unique identifiers because v_s is the sub-store of s. Since the proof is basing on the statement that holds for any arbitrary s, v, and $id :: r$, then $bind(v_s, r, v)$ always returns a store that has unique identifiers. Thus, it is valid to state that $s' = \langle id, bind(v_s, r, v) \rangle :: s_p$ has unique identifiers and $\forall s_j \subset_{\mathcal{S}} s'$ also has unique identifiers.
  - When $v_s \notin \mathcal{S}$, then it can be ignored since it always produces $\boldsymbol{err}_1$.
- Second is when $s = \langle id_s, v_s \rangle :: s_p$. Since this will match the fifth equation of definition 3.14, then it must be shown that $s' = \langle id_s, v_s \rangle :: bind(s_p, id :: r, v)$ has

unique identifiers. Based on the premise, no element in s_p whose identifier is equal to id. Since the proof is basing on the statement that holds for any arbitrary s, v, and $id :: r$, then $bind(s_p, id :: r, v)$ always returns a store that has unique identifiers, each of which is not equal to id. This, it is valid to state that $s' = \langle id_s, v_s \rangle :: bind(s_p, id :: r, v)$ has unique identifiers and $\forall s_j \subset_{\mathcal{S}} s'$ also has unique identifiers.

Since the basis and the inductive step have been performed, then by mathematical induction, the statement in proposition 3.18 holds. □

Proposition 3.21. If function *resolvelink* is used to resolve a cyclic link reference, then it will produce an error.

Proof. The proof is by contradiction. Assume that we have a series of cyclic link reference: $\langle link, r_1 \rangle, ..., \langle link, r_n \rangle, \langle link, r_1 \rangle$ where $resolve(s, ns, r_i) = \langle link, r_{i+1} \rangle$ for $i < n$ and $resolve(s, ns, r_n) = \langle link, r_1 \rangle$, $r_i, ns \in \mathcal{R}, s \in \mathcal{S}$. We assume that $resolvelink(s, \langle link, r_1 \rangle)$ will not produce an error. Based on definition 3.20, the accumulator of the next recursive call is the union of the current accumulator and the current link reference. Thus, the series of the value of accumulator (acc) inside function $\overline{resolvelink}$ in the evaluation of $resolvelink(s, \langle link, r_1 \rangle)$ will be:
$r = r_1, acc = \{\}$
$r = r_2, acc = \{r_1\}$
...
$r = r_n, acc = \{r_j | j \in 1..(n-1)\}$
$r = r_1, acc = \{r_j | j \in 1..n\}$
Above shows that at the last run, when $r = r_1$ then $r \in acc$, which produces $\boldsymbol{err}_4$ due to the first conditional branch of $\overline{resolvelink}$, a contradiction to the assumption. Thus, it is shown that the statement holds. □

Appendix B

SFP Language

B.1 Concrete Syntax

The following is the concrete syntax of SFP language.

SFP	::=	SC
SC	::=	`schema` S SC | `global` G SC | A SC | ε
B	::=	`global` G B | A B | ε
A	::=	`def` R Ac | R TV V
P	::=	(`extends`)$^{?}$ R | `{` B `}`
PS	::=	P (`,` P)* | ε
V	::=	= BV eos | LR eos | SS_o PS
R	::=	I (`.` I)*
DR	::=	R
LR	::=	R
Vec	::=	`[` (BV (`,` BV)* | ε) `]`
Null	::=	`null`
Bool	::=	`true` | `false`
BV	::=	Bool | Num | Str | DR | Vec | Null
S	::=	I SS_s `{` B `}`
SS_o	::=	`isa` SS | ε
SS_s	::=	`extends` SS | ε
TV	::=	`:` T | ε
eos	::=	`;` | `"\n"`

```
T   ::=  [] τ | * τ | τ
τ   ::=  bool | num | str | obj | I
G   ::=  And
And ::=  { (CS)* }
Or  ::=  ( (CS)* )
CS  ::=  Eq | Ne | Not | Im | And | Or | ML
Eq  ::=  R = BV eos
Ne  ::=  R != BV eos
Im  ::=  if And then And
Not ::=  not CS
ML  ::=  R in Vec eos
Ac  ::=  ( Pa ) { Co Cd Ef }
Pa  ::=  (I : T)*
Co  ::=  cost = Num eos | ε
Cd  ::=  And | ε
Ef  ::=  { (R = BV eos)+ }
```

Appendix C

Examples of System Configuration Task in SFP and PDDL

C.1 System-A

C.1.1 Cloud Deployment Scenario

C.1.1.1 SFP

The followings are the resource models, the current state, the desired state and the global constraints of a configuration task in SFP for deploying system-A (from scratch) that has 2 subsystems, each of which has 2 application services.

```
1  // file: model.sfp
2  schema Cloud {
3    def create_vm(vm: VM) {
4      condition { vm.in_cloud = null; vm.running = false; }
5      effect    { vm.in_cloud = this; }
6    }
7  }
8  schema VM {
9    in_cloud : *Cloud = null; running = false;
10   def start {
11     condition { this.in_cloud != null; this.running = false; }
12     effect    { this.running = true; }
13   }
14   def stop {
15     condition { this.in_cloud != null; this.running = true; }
16     effect    { this.running = false; }
17   }
18 }
19 schema Service {
20   installed = false;
21   running = false;
22   def start {
```

```
23      condition { this.installed = true; this.running = false;  }
24      effect    { this.running = true; }
25    }
26    def stop {
27      condition { this.installed = true; this.running = true; }
28      effect    { this.running = false; }
29    }
30    def install {
31      condition { this.installed = false; this.running = false; }
32      effect    { this.installed = true; }
33    }
34    def uninstall {
35      condition { this.installed = true; this.running = false; }
36      effect    { this.installed = false; }
37    }
38  }
39  schema LoadBalancer extends Service { }
40  schema AppService extends Service { }
```

```
1   // SFP current state
2   include "model.sfp";
3   main {
4     cloud0 isa Cloud { }
5     vm0 isa VM {
6       in_cloud = null;
7       lb isa LoadBalancer { }
8     }
9     vm0_0_0 isa VM {
10      in_cloud = null;
11      app isa AppService { }
12    }
13    vm0_0_1 extends vm0_0_0
14    vm0_1_0 extends vm0_0_0
15    vm0_1_1 extends vm0_0_0
16  }
```

```
1   // SFP desired state
2   include "model.sfp";
3   main {
4     cloud0 isa Cloud { }
5     vm0 isa VM {
6       in_cloud = cloud0;
7       running = true;
8       lb isa LoadBalancer { installed = true; running = true; }
9     }
10    vm0_0_0 isa VM {
11      in_cloud = cloud0;
12      running = true;
13      app isa AppService { installed = true; running = true; }
14    }
15    vm0_0_1 extends vm0_0_0
16    vm0_1_0 extends vm0_0_0
17    vm0_1_1 extends vm0_0_0
18    global constraint { // SFP global constraints
19      if vm0.lb.installed = true; then vm0.running = true;
```

```
20      if vm0_0_0.app.installed = true; then vm0_0_0.running = true;
21      if vm0.lb.running = true; then vm0_0_0.app.running = true;
22      if vm0_0_1.app.installed = true; then vm0_0_1.running = true;
23      if vm0.lb.running = true; then vm0_0_1.app.running = true;
24      if vm0_1_0.app.installed = true; then vm0_1_0.running = true;
25      if vm0_0_0.app.running = true; then vm0_1_0.app.running = true;
26      if vm0_1_1.app.installed = true; then vm0_1_1.running = true;
27      if vm0_0_1.app.running = true; then vm0_1_1.app.running = true;
28    }
29  }
```

C.1.1.2 PDDL

The following are the planning domain and problem in PDDL for deploying system-A (from scratch) that has 2 subsystems, each of which has 2 application services.

```
1  // PDDL Domain
2  (define (domain SystemA)
3    (:requirements :strips :typing :adl)
4    (:types runnable - object
5          cloud vm service - runnable
6          loadbalancer appservice - service)
7    (:predicates
8          (running ?r - runnable)
9          (in_cloud ?v - vm ?c - cloud)
10         (installed ?s - service))
11   (:action create-vm
12    :parameters (?c - cloud ?v - vm)
13    :precondition (and (not (exists (?cx - cloud)
14                        (in_cloud ?v ?cx))))
15    :effect (and (in_cloud ?v ?c)))
16   (:action start-vm
17    :parameters (?v - vm)
18    :precondition (and (exists (?c - cloud) (in_cloud ?v ?c))
19                        (not (running ?v)))
20    :effect (and (running ?v)))
21   (:action stop-vm
22    :parameters (?v - vm)
23    :precondition (and (exists (?c - cloud) (in_cloud ?v ?c))
24                        (running ?v))
25    :effect (and (not (running ?v))))
26   (:action install-service
27    :parameters (?s - service)
28    :precondition (and (not (installed ?s)) (not (running ?s)))
29    :effect (and (installed ?s)))
30   (:action uninstall-service
31    :parameters (?s - service)
32    :precondition (and (installed ?s) (not (running ?s)))
33    :effect (and (not (installed ?s))))
34   (:action start-service
35    :parameters (?s - service)
36    :precondition (and (installed ?s) (not (running ?s)))
37    :effect (and (running ?s)))
38   (:action stop-service
```

```
    :parameters (?s - service)
    :precondition (and (installed ?s) (running ?s))
    :effect (and (not (running ?s))))
)
```

```
;; PDDL task for 2 subsystems, each has 2 services
(define (problem p22)
  (:domain SystemA)
  (:objects
    cloud0 - cloud
    vm0 vm0_0_0 vm0_0_1 vm0_1_0 vm0_1_1 - vm
    vm0_lb - service
    vm0_0_0_app vm0_0_1_app vm0_1_0_app vm0_1_1_app - service )
  ;; current state
  (:init (running cloud0) )
  ;; desired state
  (:goal (and
    (in_cloud vm0 cloud0)     (running vm0)
    (installed vm0_lb)        (running vm0_lb)
    (in_cloud vm0_0_0 cloud0) (running vm0_0_0)
    (installed vm0_0_0_app)   (running vm0_0_0_app)
    (in_cloud vm0_0_1 cloud0) (running vm0_0_1)
    (installed vm0_0_1_app)   (running vm0_0_1_app)
    (in_cloud vm0_1_0 cloud0) (running vm0_1_0)
    (installed vm0_1_0_app)   (running vm0_1_0_app)
    (in_cloud vm0_1_1 cloud0) (running vm0_1_1)
    (installed vm0_1_1_app)   (running vm0_1_1_app) ))
  ;; global constraints
  (:constraints (and
    (always (imply (installed vm0_lb) (running vm0)))
    (always (imply (installed vm0_0_0_app) (running vm0_0_0)))
    (always (imply (running vm0_lb) (running vm0_0_0_app)))
    (always (imply (installed vm0_0_1_app) (running vm0_0_1)))
    (always (imply (running vm0_lb) (running vm0_0_1_app)))
    (always (imply (installed vm0_1_0_app) (running vm0_1_0)))
    (always (imply (running vm0_0_0_app) (running vm0_1_0_app)))
    (always (imply (installed vm0_1_1_app) (running vm0_1_1)))
    (always (imply (running vm0_0_1_app) (running vm0_1_1_app)))
  ))
)
```

C.1.2 Cloud Burst Scenario

C.1.2.1 SFP

The following are the resource models, the current state, the desired state and the global constraints of a configuration task in SFP for migrating system-A from one (`cloud0`) to another cloud (`cloud1`). In this case, system-A has 2 subsystems, each has 2 application services, and one load balancer.

```
// file: model.sfp
```

```
schema Runnable {
  running = false;
  def start {
    condition { this.running = false; }
    effect    { this.running = true;  }
  }
  def stop {
    condition { this.running = true;  }
    effect    { this.running = false; }
  }
}
schema Cloud {
  def migrate(vm: VM, target: Cloud) {
    condition { vm.in_cloud = this; vm.running = false; }
    effect    { vm.in_cloud = target; }
  }
}
schema VM extends Runnable {
  in_cloud: *Cloud = null;
}
schema LoadBalancer extends Runnable { }
schema AppService extends Runnable { }
schema Client {
  refer: *LoadBalancer = null;
  def redirect(s: LoadBalancer) {
    condition { s.running = true; }
    effect    { this.refer = s;   }
  }
}
```

```
include "model.sfp";
// current state
main {
  cloud0 isa Cloud { }
  cloud1 isa Cloud { }
  vm0 isa VM {
    in_cloud = cloud0; // on cloud0
    running = true;
    web isa LoadBalancer { running = true; }
  }
  vm0_0_0 isa VM {
    in_cloud = cloud0; // on cloud0
    running = true;
    app isa AppService { running = true; }
  }
  vm0_0_1 extends vm0_0_0
  vm0_1_0 extends vm0_0_0
  vm0_1_1 extends vm0_0_0
  vm1 isa VM {
    in_cloud = cloud0; // on cloud0
    web isa LoadBalancer { }
  }
  vm1_0_0 isa VM {
    in_cloud = cloud0; // on cloud0
    app isa AppService { }
  }
```

```
    vm1_0_1 extends vm1_0_0
    vm1_1_0 extends vm1_0_0
    vm1_1_1 extends vm1_0_0
    client0 isa Client { refer = vm0.web; }
}
```

```
include "model.sfp";
// desired state
main {
  cloud0 isa Cloud { }
  cloud1 isa Cloud { }
  vm0 isa VM {
    in_cloud = cloud1; // move to cloud1
    running = true;
    web isa LoadBalancer { running = true; }
  }
  vm0_0_0 isa VM {
    in_cloud = cloud1; // move to cloud1
    running = true;
    app isa AppService { running = true; }
  }
  vm0_0_1 extends vm0_0_0
  vm0_1_0 extends vm0_0_0
  vm0_1_1 extends vm0_0_0
  vm1 isa VM {
    in_cloud = cloud0; // stay on cloud0
    web isa LoadBalancer { }
  }
  vm1_0_0 isa VM {
    in_cloud = cloud0; // stay on cloud0
    app isa AppService { }
  }
  vm1_0_1 extends vm1_0_0
  vm1_1_0 extends vm1_0_0
  vm1_1_1 extends vm1_0_0
  client0 isa Client { refer = vm0.web; }
  global constraint {
    if vm0.web.running = true; then vm0.running = true;
    if vm1.web.running = true; then vm1.running = true;
    if vm0_0_0.app.running = true; then vm0_0_0.running = true;
    if vm0.web.running = true; then vm0_0_0.app.running = true;
    if vm1_0_0.app.running = true; then vm1_0_0.running = true;
    if vm1.web.running = true; then vm1_0_0.app.running = true;
    if vm0_0_1.app.running = true; then vm0_0_1.running = true;
    if vm0.web.running = true; then vm0_0_1.app.running = true;
    if vm1_0_1.app.running = true; then vm1_0_1.running = true;
    if vm1.web.running = true; then vm1_0_1.app.running = true;
    if vm0_1_0.app.running = true; then vm0_1_0.running = true;
    if vm0_0_0.app.running = true; then vm0_1_0.app.running = true;
    if vm1_1_0.app.running = true; then vm1_1_0.running = true;
    if vm1_0_0.app.running = true; then vm1_1_0.app.running = true;
    if vm0_1_1.app.running = true; then vm0_1_1.running = true;
    if vm0_0_1.app.running = true; then vm0_1_1.app.running = true;
    if vm1_1_1.app.running = true; then vm1_1_1.running = true;
    if vm1_0_1.app.running = true; then vm1_1_1.app.running = true;
    if client0.refer = vm0.web; then vm0.web.running = true;
```

```
51      if client0.refer = vm1.web; then vm1.web.running = true;
52    }
53  }
```

C.1.2.2 PDDL

The following are the planning domain and problem in PDDL for migrating system-A from one (cloud0) to another cloud (cloud1). In this case, system-A has 2 subsystems, each has 2 application services, and one load balancer.

```
(define (domain CloudDeploy)
  (:requirements :strips :typing :adl)
  (:types runnable client - object
    cloud vm service - runnable
    loadbalancer appservice - service )
  (:predicates
    (running ?r - runnable)
    (in_cloud ?v - vm ?c - cloud)
    (refer ?c - client ?lb - loadbalancer) )
  (:action redirect
   :parameters (?from ?to - loadbalancer ?c - client)
   :precondition (and (running ?to) (refer ?c ?from))
   :effect (and (refer ?c ?to) (not (refer ?c ?from))))
  (:action migrate
   :parameters (?from ?to - cloud ?v - vm)
   :precondition (and (not (running ?v)) (in_cloud ?v ?from)
                      (not (in_cloud ?v ?to)))
   :effect (and (not (in_cloud ?v ?from)) (in_cloud ?v ?to)))
  (:action start-vm
   :parameters (?v - vm)
   :precondition (and (exists (?c - cloud) (in_cloud ?v ?c))
                      (not (running ?v)))
   :effect (and (running ?v)))
  (:action stop-vm
   :parameters (?v - vm)
   :precondition (and (exists (?c - cloud) (in_cloud ?v ?c))
                      (running ?v))
   :effect (and (not (running ?v))))
  (:action start-service
   :parameters (?s - service)
   :precondition (and (not (running ?s)))
   :effect (and (running ?s)))
  (:action stop-service
   :parameters (?s - service)
   :precondition (and (running ?s))
   :effect (and (not (running ?s))))
)
```

```
(define (problem pcd)
(:domain CloudDeploy)
(:objects
  cloud0 cloud1 - cloud
  vm0 vm1 vm0_0_0 vm1_0_0 vm0_0_1 vm1_0_1
```

```
    vm0_1_0 vm1_1_0 vm0_1_1 vm1_1_1 - vm
    vm0_lb vm1_lb - loadbalancer
    vm0_0_0_app vm1_0_0_app vm0_0_1_app vm1_0_1_app vm0_1_0_app
    vm1_1_0_app vm0_1_1_app vm1_1_1_app - service
    client0 - client
)
(:init
    (in_cloud vm0 cloud0)      (running vm0) (running vm0_lb)
    (in_cloud vm1 cloud0)      (in_cloud vm0_0_0 cloud0)
    (running vm0_0_0)          (running vm0_0_0_app)
    (in_cloud vm1_0_0 cloud0) (in_cloud vm0_0_1 cloud0)
    (running vm0_0_1)          (running vm0_0_1_app)
    (in_cloud vm1_0_1 cloud0) (in_cloud vm0_1_0 cloud0)
    (running vm0_1_0)          (running vm0_1_0_app)
    (in_cloud vm1_1_0 cloud0) (in_cloud vm0_1_1 cloud0)
    (running vm0_1_1)          (running vm0_1_1_app)
    (in_cloud vm1_1_1 cloud0) (refer client0 vm0_lb)
)
(:goal (and
    (in_cloud vm0 cloud1)      (running vm0)
    (running vm0_lb)           (in_cloud vm1 cloud0)
    (not (running vm1))        (not (running vm1_lb))
    (in_cloud vm0_0_0 cloud1) (running vm0_0_0)
    (running vm0_0_0_app)      (in_cloud vm1_0_0 cloud0)
    (not (running vm1_0_0))    (not (running vm1_0_0_app))
    (in_cloud vm0_0_1 cloud1) (running vm0_0_1)
    (running vm0_0_1_app)      (in_cloud vm1_0_1 cloud0)
    (not (running vm1_0_1))    (not (running vm1_0_1_app))
    (in_cloud vm0_1_0 cloud1) (running vm0_1_0)
    (running vm0_1_0_app)      (in_cloud vm1_1_0 cloud0)
    (not (running vm1_1_0))    (not (running vm1_1_0_app))
    (in_cloud vm0_1_1 cloud1) (running vm0_1_1)
    (running vm0_1_1_app)      (in_cloud vm1_1_1 cloud0)
    (not (running vm1_1_1))    (not (running vm1_1_1_app))
    (refer client0 vm0_lb) ))
(:constraints (and
    (always (imply (running vm0_lb) (running vm0)))
    (always (imply (running vm1_lb) (running vm1)))
    (always (imply (running vm0_0_0_app) (running vm0_0_0)))
    (always (imply (running vm0_lb) (running vm0_0_0_app)))
    (always (imply (running vm1_0_0_app) (running vm1_0_0)))
    (always (imply (running vm1_lb) (running vm1_0_0_app)))
    (always (imply (running vm0_0_1_app) (running vm0_0_1)))
    (always (imply (running vm0_lb) (running vm0_0_1_app)))
    (always (imply (running vm1_0_1_app) (running vm1_0_1)))
    (always (imply (running vm1_lb) (running vm1_0_1_app)))
    (always (imply (running vm0_1_0_app) (running vm0_1_0)))
    (always (imply (running vm0_0_0_app) (running vm0_1_0_app)))
    (always (imply (running vm1_1_0_app) (running vm1_1_0)))
    (always (imply (running vm1_0_0_app) (running vm1_1_0_app)))
    (always (imply (running vm0_1_1_app) (running vm0_1_1)))
    (always (imply (running vm0_0_1_app) (running vm0_1_1_app)))
    (always (imply (running vm1_1_1_app) (running vm1_1_1)))
    (always (imply (running vm1_0_1_app) (running vm1_1_1_app)))
    (always (imply (refer client0 vm0_lb) (running vm0_lb)))
    (always (imply (refer client0 vm1_lb) (running vm1_lb))) ))
```

```
62 )
```

C.2 System-B

C.2.1 Cloud Deployment Scenario

C.2.1.1 SFP

The following are the resource models, the current state, the desired state and the global constraints of a configuration task in SFP for deploying system-B (from scratch) that has 2 layers, each of which has 2 application services.

```
1  schema Cloud {
2    def create_vm(vm: VM) {
3      condition { vm.in_cloud = null; vm.running = false; }
4      effect    { vm.in_cloud = this; }
5    }
6  }
7  schema VM {
8    in_cloud: *Cloud = null;
9    running = false;
10   def start {
11     condition { this.in_cloud != null; this.running = false; }
12     effect    { this.running = true; }
13   }
14   def stop {
15     condition { this.in_cloud != null; this.running = true; }
16     effect    { this.running = false; }
17   }
18 }
19 schema Service {
20   installed = false;
21   running = false;
22   def start {
23     condition { this.installed = true; this.running = false; }
24     effect    { this.running = true; }
25   }
26   def stop {
27     condition { this.installed = true; this.running = true; }
28     effect    { this.running = false; }
29   }
30   def install {
31     condition { this.installed = false; this.running = false; }
32     effect    { this.installed = true; }
33   }
34   def uninstall {
35     condition { this.installed = true; this.running = false; }
36     effect    { this.installed = false; }
37   }
38 }
39 schema LoadBalancer extends Service { }
```

```
schema MainLoadBalancer extends LoadBalancer { }
schema AppService extends Service { }
```

```
include "model.sfp";
// current state
main {
  cloud0 isa Cloud { }
  vm0_0 isa VM {
    in_cloud = null;
    lb isa MainLoadBalancer { }
  }
  vm0_0_0 isa VM {
    in_cloud = null;
    app isa AppService { }
  }
  vm0_0_1 extends vm0_0_0
  vm0_1 isa VM {
    in_cloud = null;
    lb isa LoadBalancer { }
  }
  vm0_1_0 extends vm0_0_0
  vm0_1_1 extends vm0_0_0
}
```

```
include "model.sfp";
// desired state
main {
  cloud0 isa Cloud { }
  vm0_0 isa VM {
    in_cloud = cloud0;
    running = true;
    lb isa MainLoadBalancer { installed = true; running = true; }
  }
  vm0_0_0 isa VM {
    in_cloud = cloud0;
    running = true;
    app isa AppService { installed = true; running = true; }
  }
  vm0_0_1 extends vm0_0_0
  vm0_1 isa VM {
    in_cloud = cloud0;
    running = true;
    lb isa LoadBalancer { installed = true; running = true; }
  }
  vm0_1_0 extends vm0_0_0
  vm0_1_1 extends vm0_0_0
  global constraint { // global constraints
    if vm0_0.lb.installed = true; then vm0_0.running = true;
    if vm0_0.lb.running = true; then vm0_0_0.app.running = true;
    if vm0_0_0.app.installed = true; then vm0_0_0.running = true;
    if vm0_0.lb.running = true; then vm0_0_1.app.running = true;
    if vm0_0_1.app.installed = true; then vm0_0_1.running = true;
    if vm0_1.lb.installed = true; then vm0_1.running = true;
    if vm0_0_0.app.running = true; then vm0_1.lb.running = true;
    if vm0_1.lb.running = true; then vm0_1_0.app.running = true;
```

```
32      if vm0_1_0.app.installed = true; then vm0_1_0.running = true;
33      if vm0_0_1.app.running = true; then vm0_1.lb.running = true;
34      if vm0_1.lb.running = true; then vm0_1_1.app.running = true;
35      if vm0_1_1.app.installed = true; then vm0_1_1.running = true;
36    }
37  }
```

C.2.1.2 PDDL

The following are the planning domain and problem in PDDL for deploying system-B (from scratch) that has 2 layers, each of which has 2 application services.

```
1  (define (domain CloudDeploy)
2    (:requirements :strips :typing :adl)
3    (:types runnable - object
4      cloud vm service - runnable
5      loadbalancer appservice - service)
6    (:predicates
7      (running ?r - runnable)
8      (in_cloud ?v - vm ?c - cloud)
9      (installed ?s - service))
10   (:action create-vm
11    :parameters (?c - cloud ?v - vm)
12    :precondition (and (not (exists (?cx - cloud)
13                      (in_cloud ?v ?cx))))
14    :effect (and (in_cloud ?v ?c)))
15   (:action start-vm
16    :parameters (?v - vm)
17    :precondition (and (exists (?c - cloud) (in_cloud ?v ?c))
18                      (not (running ?v)))
19    :effect (and (running ?v)))
20   (:action stop-vm
21    :parameters (?v - vm)
22    :precondition (and (exists (?c - cloud) (in_cloud ?v ?c))
23                      (running ?v))
24    :effect (and (not (running ?v))))
25   (:action install-service
26    :parameters (?s - service)
27    :precondition (and (not (installed ?s)) (not (running ?s)))
28    :effect (and (installed ?s)))
29   (:action uninstall-service
30    :parameters (?s - service)
31    :precondition (and (installed ?s) (not (running ?s)))
32    :effect (and (not (installed ?s))))
33   (:action start-service
34    :parameters (?s - service)
35    :precondition (and (installed ?s) (not (running ?s)))
36    :effect (and (running ?s)))
37   (:action stop-service
38    :parameters (?s - service)
39    :precondition (and (installed ?s) (running ?s))
40    :effect (and (not (running ?s))))
41 )
```

```
(define (problem pcd)
  (:domain CloudDeploy)
  (:objects
    cloud0 - cloud
    vm0_0 vm0_0_0 vm0_0_1 vm0_1 vm0_1_0 vm0_1_1 - vm
    vm0_0_lb vm0_1_lb - loadbalancer
    vm0_0_0_app vm0_0_1_app vm0_1_0_app vm0_1_1_app - service
  )
  (:init (running cloud0))
  (:goal (and
    (in_cloud  vm0_0 cloud0)   (running vm0_0)
    (installed vm0_0_lb)       (running vm0_0_lb)
    (in_cloud  vm0_0_0 cloud0) (running vm0_0_0)
    (installed vm0_0_0_app)    (running vm0_0_0_app)
    (in_cloud  vm0_0_1 cloud0) (running vm0_0_1)
    (installed vm0_0_1_app)    (running vm0_0_1_app)
    (in_cloud  vm0_1 cloud0)   (running vm0_1)
    (installed vm0_1_lb)       (running vm0_1_lb)
    (in_cloud  vm0_1_0 cloud0) (running vm0_1_0)
    (installed vm0_1_0_app)    (running vm0_1_0_app)
    (in_cloud  vm0_1_1 cloud0) (running vm0_1_1)
    (installed vm0_1_1_app)    (running vm0_1_1_app) ))
  (:constraints (and
    (always (imply (installed vm0_0_lb) (running vm0_0)))
    (always (imply (running vm0_0_lb) (running vm0_0_0_app)))
    (always (imply (installed vm0_0_0_app) (running vm0_0_0)))
    (always (imply (running vm0_0_lb) (running vm0_0_1_app)))
    (always (imply (installed vm0_0_1_app) (running vm0_0_1)))
    (always (imply (installed vm0_1_lb) (running vm0_1)))
    (always (imply (running vm0_0_0_app) (running vm0_1_lb)))
    (always (imply (running vm0_1_lb) (running vm0_1_0_app)))
    (always (imply (installed vm0_1_0_app) (running vm0_1_0)))
    (always (imply (running vm0_0_1_app) (running vm0_1_lb)))
    (always (imply (running vm0_1_lb) (running vm0_1_1_app)))
    (always (imply (installed vm0_1_1_app) (running vm0_1_1)))
  ))
)
```

C.2.2 Cloud Burst Scenario

C.2.2.1 SFP

The following are the resource models, the current state, the desired state and the global constraints of a configuration task in SFP for migrating system-B from one (`cloud0`) to another cloud (`cloud1`). In this case, system-B has 2 layers, each has 2 application services.

```
// file: model.sfp
schema Runnable {
  running = false;
  def start {
```

```
    condition { this.running = false; }
    effect    { this.running = true;  }
  }
  def stop {
    condition { this.running = true;  }
    effect    { this.running = false; }
  }
}
schema Cloud {
  def migrate(vm: VM, target: Cloud) {
    condition { vm.in_cloud = this; vm.running = false; }
    effect    { vm.in_cloud = target; }
  }
}
schema VM extends Runnable {
  in_cloud: *Cloud = null;
}
schema LoadBalancer extends Runnable { }
schema MainLoadBalancer extends LoadBalancer { }
schema AppService extends Runnable { }
schema Client {
  refer: *MainLoadBalancer = null;
  def redirect(s: MainLoadBalancer) {
    condition { s.running = true; }
    effect    { this.refer = s;   }
  }
}
```

```
include "model.sfp";
// current state
main {
  cloud0 isa Cloud { }
  cloud1 isa Cloud { }
  // main system
  vm0_0 isa VM {
    in_cloud = cloud0; // on the first cloud
    running = true;
    lb isa MainLoadBalancer { running = true; }
  }
  vm0_0_0 isa VM {
    in_cloud = cloud0; // on the first cloud
    running = true;
    app isa AppService { running = true; }
  }
  vm0_0_1 extends vm0_0_0
  vm0_1 isa VM {
    in_cloud = cloud0; // on the first cloud
    running = true;
    lb isa LoadBalancer { running = true; }
  }
  vm0_1_0 extends vm0_0_0
  vm0_1_1 extends vm0_0_0
  // backup system
  vm1_0 isa VM {
    in_cloud = cloud0;
    lb isa MainLoadBalancer { }
```

```
  }
  vm1_0_0 isa VM {
    in_cloud = cloud0;
    app isa AppService { }
  }
  vm1_0_1 extends vm1_0_0
  vm1_1 isa VM {
    in_cloud = cloud0;
    lb isa LoadBalancer { }
  }
  vm1_1_0 extends vm1_0_0
  vm1_1_1 extends vm1_0_0
  client0 isa Client { refer = vm0_0.lb; }
}
```

```
include "model.sfp";
// desired state
main {
  cloud0 isa Cloud { }
  cloud1 isa Cloud { }
  // main system
  vm0_0 isa VM {
    in_cloud = cloud1; // migrate to cloud1
    running = true;
    lb isa MainLoadBalancer { running = true; }
  }
  vm0_0_0 isa VM {
    in_cloud = cloud1; // migrate to cloud1
    running = true;
    app isa AppService { running = true; }
  }
  vm0_0_1 extends vm0_0_0
  vm0_1 isa VM {
    in_cloud = cloud1; // migrate to cloud1
    running = true;
    lb isa LoadBalancer { running = true; }
  }
  vm0_1_0 extends vm0_0_0
  vm0_1_1 extends vm0_0_0
  // backup system
  vm1_0 isa VM {
    in_cloud = cloud0;
    lb isa MainLoadBalancer { }
  }
  vm1_0_0 isa VM {
    in_cloud = cloud0;
    app isa AppService { }
  }
  vm1_0_1 extends vm1_0_0
  vm1_1 isa VM {
    in_cloud = cloud0;
    lb isa LoadBalancer { }
  }
  vm1_1_0 extends vm1_0_0
  vm1_1_1 extends vm1_0_0
  client0 isa Client { refer = vm0_0.lb; }
```

```
42   global constraint {
43     if vm0_0.lb.running = true; then vm0_0.running = true;
44     if vm1_0.lb.running = true; then vm1_0.running = true;
45     if vm0_0.lb.running = true; then vm0_0_0.app.running = true;
46     if vm1_0.lb.running = true; then vm1_0_0.app.running = true;
47     if vm0_0_0.app.running = true; then vm0_0_0.running = true;
48     if vm1_0_0.app.running = true; then vm1_0_0.running = true;
49     if vm0_0.lb.running = true; then vm0_0_1.app.running = true;
50     if vm1_0.lb.running = true; then vm1_0_1.app.running = true;
51     if vm0_0_1.app.running = true; then vm0_0_1.running = true;
52     if vm1_0_1.app.running = true; then vm1_0_1.running = true;
53     if vm0_1.lb.running = true; then vm0_1.running = true;
54     if vm1_1.lb.running = true; then vm1_1.running = true;
55     if vm0_1.lb.running = true; then vm0_1_0.app.running = true;
56     if vm1_1.lb.running = true; then vm1_1_0.app.running = true;
57     if vm0_1_0.app.running = true; then vm0_1_0.running = true;
58     if vm1_1_0.app.running = true; then vm1_1_0.running = true;
59     if vm0_0_0.app.running = true; then vm0_1.lb.running = true;
60     if vm1_0_0.app.running = true; then vm1_1.lb.running = true;
61     if vm0_1.lb.running = true; then vm0_1_1.app.running = true;
62     if vm1_1.lb.running = true; then vm1_1_1.app.running = true;
63     if vm0_1_1.app.running = true; then vm0_1_1.running = true;
64     if vm1_1_1.app.running = true; then vm1_1_1.running = true;
65     if vm0_0_1.app.running = true; then vm0_1.lb.running = true;
66     if vm1_0_1.app.running = true; then vm1_1.lb.running = true;
67     if client0.refer = vm0_0.lb; then vm0_0.lb.running = true;
68     if client0.refer = vm1_0.lb; then vm1_0.lb.running = true;
69   }
70 }
```

C.2.2.2 PDDL

The following are the planning domain and problem in PDDL for migrating system-B from one (`cloud0`) to another cloud (`cloud1`). In this case, system-B has 2 layers, each has 2 application services, and one load balancer.

```
1  (define (domain CloudDeploy)
2  (:requirements :strips :typing :adl)
3  (:types runnable client - object
4    cloud vm service - runnable
5    loadbalancer appservice - service
6    mainloadbalancer - loadbalancer
7  )
8  (:predicates
9    (running ?r - runnable)
10   (in_cloud ?v - vm ?c - cloud)
11   (refer ?c - client ?lb - mainloadbalancer)
12 )
13 (:action redirect
14  :parameters (?from ?to - mainloadbalancer ?c - client)
15  :precondition (and (running ?to) (refer ?c ?from))
16  :effect (and (refer ?c ?to) (not (refer ?c ?from))))
17 (:action migrate
```

```
 :parameters (?from ?to - cloud ?v - vm)
 :precondition (and (not (running ?v)) (in_cloud ?v ?from)
                    (not (in_cloud ?v ?to)))
 :effect (and (not (in_cloud ?v ?from)) (in_cloud ?v ?to)))
(:action start-vm
 :parameters (?v - vm)
 :precondition (and (exists (?c - cloud) (in_cloud ?v ?c))
                    (not (running ?v)))
 :effect (and (running ?v)))
(:action stop-vm
 :parameters (?v - vm)
 :precondition (and (exists (?c - cloud) (in_cloud ?v ?c))
                    (running ?v))
 :effect (and (not (running ?v))))
(:action start-service
 :parameters (?s - service)
 :precondition (and (not (running ?s)))
 :effect (and (running ?s)))
(:action stop-service
 :parameters (?s - service)
 :precondition (and (running ?s))
 :effect (and (not (running ?s))))
)
```

```
(define (problem pcd)
(:domain CloudDeploy)
(:objects
  cloud0 cloud1 - cloud
  vm0_0 vm1_0 vm0_0_0 vm1_0_0 vm0_0_1 vm1_0_1
  vm0_1 vm1_1 vm0_1_0 vm1_1_0 vm0_1_1 vm1_1_1 - vm
  vm0_0_lb vm1_0_lb - mainloadbalancer
  vm0_1_lb vm1_1_lb - loadbalancer
  vm0_0_0_app vm1_0_0_app vm0_0_1_app vm1_0_1_app
  vm0_1_0_app vm1_1_0_app vm0_1_1_app - service
  vm1_1_1_app - service
  client0 - client
)
(:init
  (in_cloud vm0_0 cloud0)   (running vm0_0)
  (running vm0_0_lb)        (in_cloud vm1_0 cloud0)
  (in_cloud vm0_0_0 cloud0) (running vm0_0_0)
  (running vm0_0_0_app)     (in_cloud vm1_0_0 cloud0)
  (in_cloud vm0_0_1 cloud0) (running vm0_0_1)
  (running vm0_0_1_app)     (in_cloud vm1_0_1 cloud0)
  (in_cloud vm0_1 cloud0)   (running vm0_1)
  (running vm0_1_lb)        (in_cloud vm1_1 cloud0)
  (in_cloud vm0_1_0 cloud0) (running vm0_1_0)
  (running vm0_1_0_app)     (in_cloud vm1_1_0 cloud0)
  (in_cloud vm0_1_1 cloud0) (running vm0_1_1)
  (running vm0_1_1_app)     (in_cloud vm1_1_1 cloud0)
  (refer client0 vm0_0_lb)
)
(:goal (and
  (in_cloud vm0_0 cloud1)   (running vm0_0)
  (running vm0_0_lb)        (in_cloud vm1_0 cloud0)
  (not (running vm1_0))     (not (running vm1_0_lb))
```

```
33    (in_cloud vm0_0_0 cloud1) (running vm0_0_0)
34    (running vm0_0_0_app)     (in_cloud vm1_0_0 cloud0)
35    (not (running vm1_0_0))   (not (running vm1_0_0_app))
36    (in_cloud vm0_0_1 cloud1) (running vm0_0_1)
37    (running vm0_0_1_app)     (in_cloud vm1_0_1 cloud0)
38    (not (running vm1_0_1))   (not (running vm1_0_1_app))
39    (in_cloud vm0_1 cloud1)   (running vm0_1)
40    (running vm0_1_lb)        (in_cloud vm1_1 cloud0)
41    (not (running vm1_1))     (not (running vm1_1_lb))
42    (in_cloud vm0_1_0 cloud1) (running vm0_1_0)
43    (running vm0_1_0_app)     (in_cloud vm1_1_0 cloud0)
44    (not (running vm1_1_0))   (not (running vm1_1_0_app))
45    (in_cloud vm0_1_1 cloud1) (running vm0_1_1)
46    (running vm0_1_1_app)     (in_cloud vm1_1_1 cloud0)
47    (not (running vm1_1_1))   (not (running vm1_1_1_app))
48    (refer client0 vm0_0_lb) ))
49  (:constraints (and
50    (always (imply (running vm0_0_lb) (running vm0_0)))
51    (always (imply (running vm1_0_lb) (running vm1_0)))
52    (always (imply (running vm0_0_lb) (running vm0_0_0_app)))
53    (always (imply (running vm1_0_lb) (running vm1_0_0_app)))
54    (always (imply (running vm0_0_0_app) (running vm0_0_0)))
55    (always (imply (running vm1_0_0_app) (running vm1_0_0)))
56    (always (imply (running vm0_0_lb) (running vm0_0_1_app)))
57    (always (imply (running vm1_0_lb) (running vm1_0_1_app)))
58    (always (imply (running vm0_0_1_app) (running vm0_0_1)))
59    (always (imply (running vm1_0_1_app) (running vm1_0_1)))
60    (always (imply (running vm0_1_lb) (running vm0_1)))
61    (always (imply (running vm1_1_lb) (running vm1_1)))
62    (always (imply (running vm0_1_lb) (running vm0_1_0_app)))
63    (always (imply (running vm1_1_lb) (running vm1_1_0_app)))
64    (always (imply (running vm0_1_0_app) (running vm0_1_0)))
65    (always (imply (running vm1_1_0_app) (running vm1_1_0)))
66    (always (imply (running vm0_0_0_app) (running vm0_1_lb)))
67    (always (imply (running vm1_0_0_app) (running vm1_1_lb)))
68    (always (imply (running vm0_1_lb) (running vm0_1_1_app)))
69    (always (imply (running vm1_1_lb) (running vm1_1_1_app)))
70    (always (imply (running vm0_1_1_app) (running vm0_1_1)))
71    (always (imply (running vm1_1_1_app) (running vm1_1_1)))
72    (always (imply (running vm0_0_1_app) (running vm0_1_lb)))
73    (always (imply (running vm1_0_1_app) (running vm1_1_lb)))
74    (always (imply (refer client0 vm0_0_lb) (running vm0_0_lb)))
75    (always (imply (refer client0 vm1_0_lb) (running vm1_0_lb)))
76  ))
77  )
```

C.3 System-C

C.3.1 Cloud Deployment Scenario

C.3.1.1 SFP

The following are the resource models, the current state, the desired state and the global constraints of a configuration task in SFP for deploying system-C (from scratch) that has 10 application services and 1 main service.

```
// file: model.sfp
schema Cloud {
  def create_vm(vm: VM) {
    condition { vm.in_cloud = null; vm.running = false; }
    effect    { vm.in_cloud = this; }
  }
}
schema VM {
  in_cloud: *Cloud = null;
  running = false;
  def start {
    condition { this.in_cloud != null; this.running = false; }
    effect    { this.running = true; }
  }
  def stop {
    condition { this.in_cloud != null; this.running = true; }
    effect    { this.running = false; }
  }
}
schema Service {
  installed = false;
  running = false;
  def start {
    condition { this.installed = true; this.running = false; }
    effect    { this.running = true; }
  }
  def stop {
    condition { this.installed = true; this.running = true; }
    effect    { this.running = false; }
  }
  def install {
    condition { this.installed = false; this.running = false; }
    effect    { this.installed = true; }
  }
  def uninstall {
    condition { this.installed = true; this.running = false; }
    effect    { this.installed = false; }
  }
}
schema AppService extends Service { }
schema MainAppService extends AppService { }
```

```
include "model.sfp";
```

```
// current state
main {
  cloud0 isa Cloud { }
  vm0 isa VM {
    in_cloud = null;
    lb isa MainAppService { }
  }
  vm0_0 isa VM {
    in_cloud = null;
    app isa AppService { }
  }
  vm0_1 extends vm0_0
  vm0_2 extends vm0_0
  vm0_3 extends vm0_0
  vm0_4 extends vm0_0
  vm0_5 extends vm0_0
  vm0_6 extends vm0_0
  vm0_7 extends vm0_0
  vm0_8 extends vm0_0
  vm0_9 extends vm0_0
}
```

```
include "model.sfp";
// desired state
main {
  cloud0 isa Cloud { }
  vm0 isa VM {
    in_cloud = cloud0;
    running = true;
    lb isa MainAppService { installed = true; running = true; }
  }
  vm0_0 isa VM {
    in_cloud = cloud0;
    running = true;
    app isa AppService { installed = true; running = true; }
  }
  vm0_1 extends vm0_0
  vm0_2 extends vm0_0
  vm0_3 extends vm0_0
  vm0_4 extends vm0_0
  vm0_5 extends vm0_0
  vm0_6 extends vm0_0
  vm0_7 extends vm0_0
  vm0_8 extends vm0_0
  vm0_9 extends vm0_0
  global constraint {
    if vm0.lb.installed = true; then vm0.running = true;
    if vm0.lb.running = true; then vm0_0.app.running = true;
    if vm0_0.app.installed = true; then vm0_0.running = true;
    if vm0_1.app.installed = true; then vm0_1.running = true;
    if vm0_2.app.installed = true; then vm0_2.running = true;
    if vm0_3.app.installed = true; then vm0_3.running = true;
    if vm0_4.app.installed = true; then vm0_4.running = true;
    if vm0_5.app.installed = true; then vm0_5.running = true;
    if vm0_6.app.installed = true; then vm0_6.running = true;
    if vm0_7.app.installed = true; then vm0_7.running = true;
```

```
35      if vm0_8.app.installed = true; then vm0_8.running = true;
36      if vm0_9.app.installed = true; then vm0_9.running = true;
37      if vm0_0.app.running = true; then vm0_1.app.running = true;
38      if vm0_0.app.running = true; then vm0_4.app.running = true;
39      if vm0_0.app.running = true; then vm0_5.app.running = true;
40      if vm0_0.app.running = true; then vm0_6.app.running = true;
41      if vm0_1.app.running = true; then vm0_8.app.running = true;
42      if vm0_1.app.running = true; then vm0_2.app.running = true;
43      if vm0_1.app.running = true; then vm0_6.app.running = true;
44      if vm0_1.app.running = true; then vm0_7.app.running = true;
45      if vm0_2.app.running = true; then vm0_9.app.running = true;
46      if vm0_2.app.running = true; then vm0_3.app.running = true;
47      if vm0_3.app.running = true; then vm0_4.app.running = true;
48      if vm0_3.app.running = true; then vm0_5.app.running = true;
49      if vm0_4.app.running = true; then vm0_8.app.running = true;
50      if vm0_4.app.running = true; then vm0_5.app.running = true;
51      if vm0_4.app.running = true; then vm0_6.app.running = true;
52      if vm0_5.app.running = true; then vm0_8.app.running = true;
53      if vm0_5.app.running = true; then vm0_9.app.running = true;
54      if vm0_5.app.running = true; then vm0_7.app.running = true;
55      if vm0_6.app.running = true; then vm0_8.app.running = true;
56      if vm0_6.app.running = true; then vm0_9.app.running = true;
57      if vm0_6.app.running = true; then vm0_7.app.running = true;
58      if vm0_7.app.running = true; then vm0_8.app.running = true;
59    }
60  }
```

C.3.1.2 PDDL

The following are the planning domain and problem in PDDL for deploying system-C (from scratch) that has 10 application services and 1 main service.

```
1  (define (domain CloudDeploy)
2    (:requirements :strips :typing :adl)
3    (:types runnable - object
4      cloud vm service - runnable
5      appservice - service
6      mainappservice - appservice)
7    (:predicates
8      (running ?r - runnable)
9      (in_cloud ?v - vm ?c - cloud)
10     (installed ?s - service))
11   (:action create-vm
12    :parameters (?c - cloud ?v - vm)
13    :precondition (and (not (exists (?cx - cloud)
14                       (in_cloud ?v ?cx))))
15    :effect (and (in_cloud ?v ?c)))
16   (:action start-vm
17    :parameters (?v - vm)
18    :precondition (and (exists (?c - cloud) (in_cloud ?v ?c))
19                       (not (running ?v)))
20    :effect (and (running ?v)))
21   (:action stop-vm
22    :parameters (?v - vm)
```

```
    :precondition (and (exists (?c - cloud) (in_cloud ?v ?c))
                       (running ?v))
    :effect (and (not (running ?v))))
  (:action install-service
    :parameters (?s - service)
    :precondition (and (not (installed ?s)) (not (running ?s)))
    :effect (and (installed ?s)))
  (:action uninstall-service
    :parameters (?s - service)
    :precondition (and (installed ?s) (not (running ?s)))
    :effect (and (not (installed ?s))))
  (:action start-service
    :parameters (?s - service)
    :precondition (and (installed ?s) (not (running ?s)))
    :effect (and (running ?s)))
  (:action stop-service
    :parameters (?s - service)
    :precondition (and (installed ?s) (running ?s))
    :effect (and (not (running ?s))))
)
```

```
(define (problem pcd)
  (:domain CloudDeploy)
  (:objects
    cloud0 - cloud
    vm0 - mainappservice
    vm0_0 vm0_1 vm0_2 vm0_3 vm0_4 vm0_5 vm0_6 vm0_7 vm0_8 vm0_9 - vm
    vm0_app vm0_0_app vm0_1_app vm0_2_app vm0_3_app vm0_4_app
    vm0_5_app vm0_6_app vm0_7_app vm0_8_app vm0_9_app - appservice)
  (:init (running cloud0))
  (:goal (and
    (in_cloud vm0 cloud0)   (running vm0)
    (installed vm0_app)     (running vm0_app)
    (in_cloud vm0_0 cloud0) (running vm0_0)
    (installed vm0_0_app)   (running vm0_0_app)
    (in_cloud vm0_1 cloud0) (running vm0_1)
    (installed vm0_1_app)   (running vm0_1_app)
    (in_cloud vm0_2 cloud0) (running vm0_2)
    (installed vm0_2_app)   (running vm0_2_app)
    (in_cloud vm0_3 cloud0) (running vm0_3)
    (installed vm0_3_app)   (running vm0_3_app)
    (in_cloud vm0_4 cloud0) (running vm0_4)
    (installed vm0_4_app)   (running vm0_4_app)
    (in_cloud vm0_5 cloud0) (running vm0_5)
    (installed vm0_5_app)   (running vm0_5_app)
    (in_cloud vm0_6 cloud0) (running vm0_6)
    (installed vm0_6_app)   (running vm0_6_app)
    (in_cloud vm0_7 cloud0) (running vm0_7)
    (installed vm0_7_app)   (running vm0_7_app)
    (in_cloud vm0_8 cloud0) (running vm0_8)
    (installed vm0_8_app)   (running vm0_8_app)
    (in_cloud vm0_9 cloud0) (running vm0_9)
    (installed vm0_9_app)   (running vm0_9_app)))
  (:constraints (and
    (always (imply (installed vm0_app) (running vm0)))
    (always (imply (running vm0_app) (running vm0_0_app)))
```

```
36      (always (imply (installed vm0_1_app) (running vm0_1)))
37      (always (imply (installed vm0_0_app) (running vm0_0)))
38      (always (imply (installed vm0_3_app) (running vm0_3)))
39      (always (imply (installed vm0_2_app) (running vm0_2)))
40      (always (imply (installed vm0_5_app) (running vm0_5)))
41      (always (imply (installed vm0_4_app) (running vm0_4)))
42      (always (imply (installed vm0_7_app) (running vm0_7)))
43      (always (imply (installed vm0_6_app) (running vm0_6)))
44      (always (imply (installed vm0_9_app) (running vm0_9)))
45      (always (imply (installed vm0_8_app) (running vm0_8)))
46      (always (imply (running vm0_1_app) (running vm0_8_app)))
47      (always (imply (running vm0_1_app) (running vm0_2_app)))
48      (always (imply (running vm0_1_app) (running vm0_7_app)))
49      (always (imply (running vm0_1_app) (running vm0_6_app)))
50      (always (imply (running vm0_0_app) (running vm0_1_app)))
51      (always (imply (running vm0_0_app) (running vm0_5_app)))
52      (always (imply (running vm0_0_app) (running vm0_4_app)))
53      (always (imply (running vm0_0_app) (running vm0_6_app)))
54      (always (imply (running vm0_3_app) (running vm0_5_app)))
55      (always (imply (running vm0_3_app) (running vm0_4_app)))
56      (always (imply (running vm0_2_app) (running vm0_9_app)))
57      (always (imply (running vm0_2_app) (running vm0_3_app)))
58      (always (imply (running vm0_5_app) (running vm0_9_app)))
59      (always (imply (running vm0_5_app) (running vm0_8_app)))
60      (always (imply (running vm0_5_app) (running vm0_7_app)))
61      (always (imply (running vm0_4_app) (running vm0_8_app)))
62      (always (imply (running vm0_4_app) (running vm0_5_app)))
63      (always (imply (running vm0_4_app) (running vm0_6_app)))
64      (always (imply (running vm0_7_app) (running vm0_8_app)))
65      (always (imply (running vm0_6_app) (running vm0_9_app)))
66      (always (imply (running vm0_6_app) (running vm0_8_app)))
67      (always (imply (running vm0_6_app) (running vm0_7_app)))
68    ))
69  )
```

C.3.2 Cloud Burst Scenario

C.3.2.1 SFP

The following are the resource models, the current state, the desired state and the global constraints of a configuration task in SFP for migrating system-C from one (`cloud0`) to another cloud (`cloud1`). In this case, system-C has 10 application services and 1 main service.

```
1  // file: model.sfp
2  schema Runnable {
3    running = false;
4    def start {
5      condition { this.running = false; }
6      effect    { this.running = true; }
7    }
8    def stop {
```

```
    condition { this.running = true; }
    effect    { this.running = false; }
  }
}
schema Cloud {
  def migrate(vm: VM, target: Cloud) {
    condition { vm.in_cloud = this; vm.running = false; }
    effect    { vm.in_cloud = target; }
  }
}
schema VM extends Runnable {
  in_cloud: *Cloud = null;
}
schema AppService extends Runnable { }
schema MainAppService extends Runnable { }
schema Client {
  refer: *MainAppService = null;
  def redirect(s: MainAppService) {
    condition { s.running = true; }
    effect    { this.refer = s;   }
  }
}
```

```
include "model.sfp"
// current state
main {
  cloud0 isa Cloud { }
  cloud1 isa Cloud { }
  // main system
  vm0 isa VM {
    in_cloud = cloud0;
    running = true;
    app isa MainAppService { running = true; }
  }
  vm0_0 isa VM {
    in_cloud = cloud0;
    running = true;
    app isa AppService { running = true; }
  }
  vm0_1 extends vm0_0
  vm0_2 extends vm0_0
  vm0_3 extends vm0_0
  vm0_4 extends vm0_0
  vm0_5 extends vm0_0
  vm0_6 extends vm0_0
  vm0_7 extends vm0_0
  vm0_8 extends vm0_0
  vm0_9 extends vm0_0
  // backup system
  vm1 isa VM {
    in_cloud = cloud0;
    app isa MainAppService { }
  }
  vm1_0 isa VM {
    in_cloud = cloud0;
    app isa AppService { }
```

```
  }
  vm1_1 extends vm1_0
  vm1_2 extends vm1_0
  vm1_3 extends vm1_0
  vm1_4 extends vm1_0
  vm1_5 extends vm1_0
  vm1_6 extends vm1_0
  vm1_7 extends vm1_0
  vm1_8 extends vm1_0
  vm1_9 extends vm1_0
  // client
  client0 isa Client { refer = vm0.app; }
}
```

```
include "model.sfp"
// desired state
main {
  cloud0 isa Cloud { }
  cloud1 isa Cloud { }
  // main system
  vm0 isa VM {
    in_cloud = cloud1; // migrate to cloud1
    running = true;
    app isa MainAppService { running = true; }
  }
  vm0_0 isa VM {
    in_cloud = cloud1; // migrate to cloud1
    running = true;
    app isa AppService { running = true; }
  }
  vm0_1 extends vm0_0
  vm0_2 extends vm0_0
  vm0_3 extends vm0_0
  vm0_4 extends vm0_0
  vm0_5 extends vm0_0
  vm0_6 extends vm0_0
  vm0_7 extends vm0_0
  vm0_8 extends vm0_0
  vm0_9 extends vm0_0
  // backup system
  vm1 isa VM {
    in_cloud = cloud0;
    app isa MainAppService { }
  }
  vm1_0 isa VM {
    in_cloud = cloud0;
    app isa AppService { }
  }
  vm1_1 extends vm1_0
  vm1_2 extends vm1_0
  vm1_3 extends vm1_0
  vm1_4 extends vm1_0
  vm1_5 extends vm1_0
  vm1_6 extends vm1_0
  vm1_7 extends vm1_0
  vm1_8 extends vm1_0
```

```
43    vm1_9 extends vm1_0
44    // client
45    client0 isa Client { refer = vm0.app; }
46    // global constraints
47    global constraint {
48      if vm0.app.running = true; then vm0.running = true;
49      if vm1.app.running = true; then vm1.running = true;
50      if vm0.app.running = true; then vm0_0.app.running = true;
51      if vm1.app.running = true; then vm1_0.app.running = true;
52      if vm0_0.app.running = true; then vm0_0.running = true;
53      if vm1_0.app.running = true; then vm1_0.running = true;
54      if vm0_1.app.running = true; then vm0_1.running = true;
55      if vm1_1.app.running = true; then vm1_1.running = true;
56      if vm0_2.app.running = true; then vm0_2.running = true;
57      if vm1_2.app.running = true; then vm1_2.running = true;
58      if vm0_3.app.running = true; then vm0_3.running = true;
59      if vm1_3.app.running = true; then vm1_3.running = true;
60      if vm0_4.app.running = true; then vm0_4.running = true;
61      if vm1_4.app.running = true; then vm1_4.running = true;
62      if vm0_5.app.running = true; then vm0_5.running = true;
63      if vm1_5.app.running = true; then vm1_5.running = true;
64      if vm0_6.app.running = true; then vm0_6.running = true;
65      if vm1_6.app.running = true; then vm1_6.running = true;
66      if vm0_7.app.running = true; then vm0_7.running = true;
67      if vm1_7.app.running = true; then vm1_7.running = true;
68      if vm0_8.app.running = true; then vm0_8.running = true;
69      if vm1_8.app.running = true; then vm1_8.running = true;
70      if vm0_9.app.running = true; then vm0_9.running = true;
71      if vm1_9.app.running = true; then vm1_9.running = true;
72      if vm0_0.app.running = true; then vm0_1.app.running = true;
73      if vm1_0.app.running = true; then vm1_1.app.running = true;
74      if vm0_0.app.running = true; then vm0_3.app.running = true;
75      if vm1_0.app.running = true; then vm1_3.app.running = true;
76      if vm0_0.app.running = true; then vm0_4.app.running = true;
77      if vm1_0.app.running = true; then vm1_4.app.running = true;
78      if vm0_0.app.running = true; then vm0_5.app.running = true;
79      if vm1_0.app.running = true; then vm1_5.app.running = true;
80      if vm0_0.app.running = true; then vm0_7.app.running = true;
81      if vm1_0.app.running = true; then vm1_7.app.running = true;
82      if vm0_0.app.running = true; then vm0_8.app.running = true;
83      if vm1_0.app.running = true; then vm1_8.app.running = true;
84      if vm0_0.app.running = true; then vm0_9.app.running = true;
85      if vm1_0.app.running = true; then vm1_9.app.running = true;
86      if vm0_1.app.running = true; then vm0_2.app.running = true;
87      if vm1_1.app.running = true; then vm1_2.app.running = true;
88      if vm0_1.app.running = true; then vm0_5.app.running = true;
89      if vm1_1.app.running = true; then vm1_5.app.running = true;
90      if vm0_2.app.running = true; then vm0_3.app.running = true;
91      if vm1_2.app.running = true; then vm1_3.app.running = true;
92      if vm0_2.app.running = true; then vm0_7.app.running = true;
93      if vm1_2.app.running = true; then vm1_7.app.running = true;
94      if vm0_3.app.running = true; then vm0_8.app.running = true;
95      if vm1_3.app.running = true; then vm1_8.app.running = true;
96      if vm0_3.app.running = true; then vm0_9.app.running = true;
97      if vm1_3.app.running = true; then vm1_9.app.running = true;
98      if vm0_4.app.running = true; then vm0_8.app.running = true;
```

```
99      if vm1_4.app.running = true; then vm1_8.app.running = true;
100     if vm0_4.app.running = true; then vm0_5.app.running = true;
101     if vm1_4.app.running = true; then vm1_5.app.running = true;
102     if vm0_5.app.running = true; then vm0_8.app.running = true;
103     if vm1_5.app.running = true; then vm1_8.app.running = true;
104     if vm0_5.app.running = true; then vm0_9.app.running = true;
105     if vm1_5.app.running = true; then vm1_9.app.running = true;
106     if vm0_5.app.running = true; then vm0_6.app.running = true;
107     if vm1_5.app.running = true; then vm1_6.app.running = true;
108     if client0.refer = vm0.app; then vm0.app.running = true;
109     if client0.refer = vm1.app; then vm1.app.running = true;
110   }
111 }
```

C.3.2.2 PDDL

The following are the planning domain and problem in PDDL for migrating system-C from one (`cloud0`) to another cloud (`cloud1`). In this case, system-C has 10 application services and 1 main service.

```
1  (define (domain CloudDeploy)
2    (:requirements :strips :typing :adl)
3    (:types runnable client - object
4      cloud vm service - runnable
5      appservice - service
6      mainappservice - appservice)
7    (:predicates
8      (running ?r - runnable)
9      (in_cloud ?v - vm ?c - cloud)
10     (refer ?c - client ?lb - mainappservice))
11   (:action redirect
12    :parameters (?from ?to - mainappservice ?c - client)
13    :precondition (and (running ?to) (refer ?c ?from))
14    :effect (and (refer ?c ?to) (not (refer ?c ?from))))
15   (:action migrate
16    :parameters (?from ?to - cloud ?v - vm)
17    :precondition (and (not (running ?v)) (in_cloud ?v ?from)
18                       (not (in_cloud ?v ?to)))
19    :effect (and (not (in_cloud ?v ?from)) (in_cloud ?v ?to)))
20   (:action start-vm
21    :parameters (?v - vm)
22    :precondition (and (exists (?c - cloud) (in_cloud ?v ?c))
23                       (not (running ?v)))
24    :effect (and (running ?v)))
25   (:action stop-vm
26    :parameters (?v - vm)
27    :precondition (and (exists (?c - cloud) (in_cloud ?v ?c))
28                       (running ?v))
29    :effect (and (not (running ?v))))
30   (:action start-service
31    :parameters (?s - service)
32    :precondition (and (not (running ?s)))
33    :effect (and (running ?s)))
```

```
  (:action stop-service
   :parameters (?s - service)
   :precondition (and (running ?s))
   :effect (and (not (running ?s))))
)
```

```
(define (problem pcd)
  (:domain CloudDeploy)
  (:objects
    cloud0 cloud1 - cloud
    vm0 vm1 vm0_0 vm1_0 vm0_1 vm1_1 vm0_2 vm1_2 vm0_3 vm1_3
    vm0_4 vm1_4 vm0_5 vm1_5 vm0_6 vm1_6 vm0_7 vm1_7 vm0_8
    vm1_8 vm0_9 vm1_9 - vm
    vm0_app vm1_app - mainappservice
    vm0_0_app vm1_0_app vm0_1_app vm1_1_app vm0_2_app
    vm1_2_app vm0_3_app vm1_3_app vm0_4_app vm1_4_app
    vm0_5_app vm1_5_app vm0_6_app vm1_6_app vm0_7_app
    vm1_7_app vm0_8_app vm1_8_app vm0_9_app vm1_9_app
    - appservice
    client0 - client
  )
  (:init
    (in_cloud vm0 cloud0)
    (running vm0)
    (running vm0_app)
    (in_cloud vm1 cloud0)
    (in_cloud vm0_0 cloud0)
    (running vm0_0)
    (running vm0_0_app)
    (in_cloud vm1_0 cloud0)
    (in_cloud vm0_1 cloud0)
    (running vm0_1)
    (running vm0_1_app)
    (in_cloud vm1_1 cloud0)
    (in_cloud vm0_2 cloud0)
    (running vm0_2)
    (running vm0_2_app)
    (in_cloud vm1_2 cloud0)
    (in_cloud vm0_3 cloud0)
    (running vm0_3)
    (running vm0_3_app)
    (in_cloud vm1_3 cloud0)
    (in_cloud vm0_4 cloud0)
    (running vm0_4)
    (running vm0_4_app)
    (in_cloud vm1_4 cloud0)
    (in_cloud vm0_5 cloud0)
    (running vm0_5)
    (running vm0_5_app)
    (in_cloud vm1_5 cloud0)
    (in_cloud vm0_6 cloud0)
    (running vm0_6)
    (running vm0_6_app)
    (in_cloud vm1_6 cloud0)
    (in_cloud vm0_7 cloud0)
    (running vm0_7)
```

```
51      (running vm0_7_app)
52      (in_cloud vm1_7 cloud0)
53      (in_cloud vm0_8 cloud0)
54      (running vm0_8)
55      (running vm0_8_app)
56      (in_cloud vm1_8 cloud0)
57      (in_cloud vm0_9 cloud0)
58      (running vm0_9)
59      (running vm0_9_app)
60      (in_cloud vm1_9 cloud0)
61      (refer client0 vm0_app)
62    )
63    (:goal (and
64      (in_cloud vm0 cloud1)
65      (running vm0)
66      (running vm0_app)
67      (in_cloud vm1 cloud0)
68      (not (running vm1))
69      (not (running vm1_app))
70      (in_cloud vm0_0 cloud1)
71      (running vm0_0)
72      (running vm0_0_app)
73      (in_cloud vm1_0 cloud0)
74      (not (running vm1_0))
75      (not (running vm1_0_app))
76      (in_cloud vm0_1 cloud1)
77      (running vm0_1)
78      (running vm0_1_app)
79      (in_cloud vm1_1 cloud0)
80      (not (running vm1_1))
81      (not (running vm1_1_app))
82      (in_cloud vm0_2 cloud1)
83      (running vm0_2)
84      (running vm0_2_app)
85      (in_cloud vm1_2 cloud0)
86      (not (running vm1_2))
87      (not (running vm1_2_app))
88      (in_cloud vm0_3 cloud1)
89      (running vm0_3)
90      (running vm0_3_app)
91      (in_cloud vm1_3 cloud0)
92      (not (running vm1_3))
93      (not (running vm1_3_app))
94      (in_cloud vm0_4 cloud1)
95      (running vm0_4)
96      (running vm0_4_app)
97      (in_cloud vm1_4 cloud0)
98      (not (running vm1_4))
99      (not (running vm1_4_app))
100     (in_cloud vm0_5 cloud1)
101     (running vm0_5)
102     (running vm0_5_app)
103     (in_cloud vm1_5 cloud0)
104     (not (running vm1_5))
105     (not (running vm1_5_app))
106     (in_cloud vm0_6 cloud1)
```

```
107       (running vm0_6)
108       (running vm0_6_app)
109       (in_cloud vm1_6 cloud0)
110       (not (running vm1_6))
111       (not (running vm1_6_app))
112       (in_cloud vm0_7 cloud1)
113       (running vm0_7)
114       (running vm0_7_app)
115       (in_cloud vm1_7 cloud0)
116       (not (running vm1_7))
117       (not (running vm1_7_app))
118       (in_cloud vm0_8 cloud1)
119       (running vm0_8)
120       (running vm0_8_app)
121       (in_cloud vm1_8 cloud0)
122       (not (running vm1_8))
123       (not (running vm1_8_app))
124       (in_cloud vm0_9 cloud1)
125       (running vm0_9)
126       (running vm0_9_app)
127       (in_cloud vm1_9 cloud0)
128       (not (running vm1_9))
129       (not (running vm1_9_app))
130       (refer client0 vm0_app)
131     ))
132     (:constraints (and
133       (always (imply (running vm0_app) (running vm0)))
134       (always (imply (running vm1_app) (running vm1)))
135       (always (imply (running vm0_app) (running vm0_0_app)))
136       (always (imply (running vm1_app) (running vm1_0_app)))
137       (always (imply (running vm0_1_app) (running vm0_1)))
138       (always (imply (running vm1_1_app) (running vm1_1)))
139       (always (imply (running vm0_0_app) (running vm0_0)))
140       (always (imply (running vm1_0_app) (running vm1_0)))
141       (always (imply (running vm0_3_app) (running vm0_3)))
142       (always (imply (running vm1_3_app) (running vm1_3)))
143       (always (imply (running vm0_2_app) (running vm0_2)))
144       (always (imply (running vm1_2_app) (running vm1_2)))
145       (always (imply (running vm0_5_app) (running vm0_5)))
146       (always (imply (running vm1_5_app) (running vm1_5)))
147       (always (imply (running vm0_4_app) (running vm0_4)))
148       (always (imply (running vm1_4_app) (running vm1_4)))
149       (always (imply (running vm0_7_app) (running vm0_7)))
150       (always (imply (running vm1_7_app) (running vm1_7)))
151       (always (imply (running vm0_6_app) (running vm0_6)))
152       (always (imply (running vm1_6_app) (running vm1_6)))
153       (always (imply (running vm0_9_app) (running vm0_9)))
154       (always (imply (running vm1_9_app) (running vm1_9)))
155       (always (imply (running vm0_8_app) (running vm0_8)))
156       (always (imply (running vm1_8_app) (running vm1_8)))
157       (always (imply (running vm0_1_app) (running vm0_2_app)))
158       (always (imply (running vm1_1_app) (running vm1_2_app)))
159       (always (imply (running vm0_1_app) (running vm0_5_app)))
160       (always (imply (running vm1_1_app) (running vm1_5_app)))
161       (always (imply (running vm0_0_app) (running vm0_1_app)))
162       (always (imply (running vm1_0_app) (running vm1_1_app)))
```

```
163       (always (imply (running vm0_0_app) (running vm0_3_app)))
164       (always (imply (running vm1_0_app) (running vm1_3_app)))
165       (always (imply (running vm0_0_app) (running vm0_5_app)))
166       (always (imply (running vm1_0_app) (running vm1_5_app)))
167       (always (imply (running vm0_0_app) (running vm0_4_app)))
168       (always (imply (running vm1_0_app) (running vm1_4_app)))
169       (always (imply (running vm0_0_app) (running vm0_7_app)))
170       (always (imply (running vm1_0_app) (running vm1_7_app)))
171       (always (imply (running vm0_0_app) (running vm0_9_app)))
172       (always (imply (running vm1_0_app) (running vm1_9_app)))
173       (always (imply (running vm0_0_app) (running vm0_8_app)))
174       (always (imply (running vm1_0_app) (running vm1_8_app)))
175       (always (imply (running vm0_3_app) (running vm0_9_app)))
176       (always (imply (running vm1_3_app) (running vm1_9_app)))
177       (always (imply (running vm0_3_app) (running vm0_8_app)))
178       (always (imply (running vm1_3_app) (running vm1_8_app)))
179       (always (imply (running vm0_2_app) (running vm0_3_app)))
180       (always (imply (running vm1_2_app) (running vm1_3_app)))
181       (always (imply (running vm0_2_app) (running vm0_7_app)))
182       (always (imply (running vm1_2_app) (running vm1_7_app)))
183       (always (imply (running vm0_5_app) (running vm0_9_app)))
184       (always (imply (running vm1_5_app) (running vm1_9_app)))
185       (always (imply (running vm0_5_app) (running vm0_8_app)))
186       (always (imply (running vm1_5_app) (running vm1_8_app)))
187       (always (imply (running vm0_5_app) (running vm0_6_app)))
188       (always (imply (running vm1_5_app) (running vm1_6_app)))
189       (always (imply (running vm0_4_app) (running vm0_8_app)))
190       (always (imply (running vm1_4_app) (running vm1_8_app)))
191       (always (imply (running vm0_4_app) (running vm0_5_app)))
192       (always (imply (running vm1_4_app) (running vm1_5_app)))
193       (always (imply (refer client0 vm0_app) (running vm0_app)))
194       (always (imply (refer client0 vm1_app) (running vm1_app)))
195     ))
196   )
```

Bibliography

[bon, 2014] (2014). BonFIRE Restfully. http://doc.bonfire-project.eu/R4.0.5/client-tools/restfully.html. Accessed: 2015-02-17.

[has, 2014] (2014). Haskell Language. http://haskell.org. Accessed: 2014-07-01.

[hpc, 2014] (2014). HP Cells. http://www.hpl.hp.com/open_innovation/cloud_collaboration/projects.html. Accessed:2014-07-01.

[oca, 2014] (2014). OCaml Language. http://ocaml.org. Accessed: 2014-07-01.

[sca, 2014] (2014). Scala Language. http://scala-lang.org. Accessed: 2014-07-01.

[yam, 2014] (2014). YAML. http://www.yaml.org. Accessed: 2014-07-01.

[Albore et al., 2009] Albore, A., Palacios, H., and Geffner, H. (2009). A translation-based approach to contingent planning. In *Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1623–1628.

[Ambite and Knoblock, 2001] Ambite, J. L. and Knoblock, C. A. (2001). Planning by rewriting. *Journal of Artificial Intelligence Research*, 15:15–207.

[Anderson, 2006] Anderson, P. (2006). *System Configuration, volume 14 of short topics in system administration*. SAGE.

[Anderson and Scobie, 2002] Anderson, P. and Scobie, A. (2002). LCFG: The next generation. In *UKUUG Winter Conference*.

[Ansible Inc., 2014] Ansible Inc. (2014). Ansible. http://ansibleworks.com. Accessed: 2014-07-01.

[Bäckström, 1994] Bäckström, C. (1994). Finding least constrained plans and optimal parallel executions is harder than we thought. In *In Proceedings of the second European Workshop on Planning*, pages 46–59. IOS Press.

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

[Baier and McIlraith, 2006] Baier, J. A. and McIlraith, S. (2006). Planning with temporally extended goals using heuristic search. In *Proceedings of the 16th International Conference on Automated Planning and Scheduling*.

[Bainomugisha et al., 2013] Bainomugisha, E., Carreton, A. L., Cutsem, T. v., Mostinckx, S., and Meuter, W. d. (2013). A survey on reactive programming. *ACM Computing Surveys*, 45(4):52:1–52:34.

[Banerjee et al., 2012] Banerjee, P., Bash, C., Friedrich, R., Goldsack, P., Huberman, B. A., Manley, J., Morell, L., Patel, C., Ranganathan, P., Trucco, M., and Veitch, A. (2012). The Future of Cloud Computing: an HP Labs perspective. http://www.hpl.hp.com/techreports/2010/HPL-2010-192.pdf (accessed: 2015-02-15).

[Bertoli et al., 2001] Bertoli, P., Cimatti, A., Roveri, M., and Traverso, P. (2001). Planning in nondeterministic domains under partial observability via symbolic model checking. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI)*, volume 2001, pages 473–478.

[Blum and Furst, 1997a] Blum, A. and Furst, M. (1997a). Fast Planning through Planning Graph Analysis. *Artificial Intelligence*, 90:281–300.

[Blum and Furst, 1997b] Blum, A. L. and Furst, M. L. (1997b). Fast planning through planning graph analysis. *Artificial intelligence*, 90(1):281–300.

[Bonet and Geffner, 2011] Bonet, B. and Geffner, H. (2011). Planning under partial observability by classical replanning: Theory and experiments. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence (IJCAI)*. Association for the Advancement of Artificial Intelligence (AAAI).

[Brafman and Shani, 2012] Brafman, R. I. and Shani, G. (2012). Replanning in domains with partial information and sensing actions. *Journal of Artificial Intelligence Research*, pages 565–600.

[Bylander, 1994] Bylander, T. (1994). The computational complexity of propositional strips planning. *Artificial Intelligence*, 69(1):165–204.

[Canonical Ltd., 2014] Canonical Ltd. (2014). Juju. http://juju.ubuntu.com. Accessed: 2014-07-01.

[Crosby, 2014] Crosby, M. D. (2014). *Multiagent Classical Planning*. PhD thesis, School of Informatics, The University of Edinburgh.

[Dantas et al., 2006] Dantas, A., Santos, F., Germoglio, G., Oliveira, M., Cirne, W., Brasileiro, F., Rafaeli, S., Saikoski, K., and Milojicic, D. (2006). An initial assessment of cddlm. In *Proceedings of the HP-OpenView University Association Workshop*.

[Delaet and Joosen, 2010] Delaet, T. and Joosen, W. (2010). A survey of system configuration tools. In *Proceedings of the 24th Large Installation System Administration Conference (LISA '10)*. Usenix Association.

[Desai et al., 2003] Desai, N., Lusk, A., Bradshaw, R., and Evard, R. (2003). BCFG: A Configuration Management Tool for Heterogeneous Environments. In *Proceedings of IEEE International Conference on Cluster Computing*. IEEE Computer Society.

[Di Cosmo et al., 2012] Di Cosmo, R., Zacchiroli, S., and Zavattaro, G. (2012). Towards a formal component model for the cloud. In *Software Engineering and Formal Methods*, pages 156–171. Springer.

[Edelkamp, 2006] Edelkamp, S. (2006). On the compilation of plan constraints and preferences. In *Proceedings of the 16th International Conference on Automated Planning and Scheduling*, pages 374–377.

[Edelkamp and Jabbar, 2008] Edelkamp, S. and Jabbar, S. (2008). MIPS-XXL: Featuring external shortest path search for sequential optimal plans and external branch-and-bound for optimal net benefit. *Sixth International Planning Competition Booklet (ICAPS 2008)*, 143.

[Edelkamp et al., 2006] Edelkamp, S., Jabbar, S., and Naizih, M. (2006). Large-scale optimal PDDL3 planning with MIPS-XXL. *5th International Planning Competition Booklet (IPC-2006)*, pages 28–30.

[El Maghraoui et al., 2006] El Maghraoui, K., Meghranjani, A., Eilam, T., Kalantar, M., and Konstantinou, A. (2006). Model driven provisioning: Bridging the gap between declarative object models and procedural provisioning tools. In *Proceedings of the ACM/IFIP/USENIX International Conference on Middleware*, pages 404–423. Springer-Verlag New York, Inc.

[Farrell, 2008] Farrell, A. (2008). Model-based orchestration. HP Labs.

[Farrell et al., 2010] Farrell, A., Prakash, S., and Rolia, J. (2010). Behavioral Signatures for Business Service Management in the Cloud. Technical report, HP Labs.

[Fikes and Nilsson, 1971] Fikes, R. and Nilsson, N. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208.

[Foundation, 2014a] Foundation, T. A. S. (2014a). Apache Hadoop. http://hadoop.apache.org. Accessed: 2014-07-01.

[Foundation, 2014b] Foundation, T. A. S. (2014b). Apache Zookeeper. http://zookeeper.apache.org. Accessed: 2014-07-01.

[Frederic Gittler, 2012] Frederic Gittler (2012). Cloud Computing Security, HP Labs G-Cloud: A Secure Cloud Infrastructure. http://www.itu.int/en/ITU-T/studygroups/com17/Documents/tutorials/2012/04-CloudSecurityHPlabs.pdf. Accessed: 2015-02-15.

[Fritz and McIlraith, 2007] Fritz, C. and McIlraith, S. A. (2007). Monitoring plan optimality during execution. In *Proceedings of the 17th International Conference on*

Automated Planning and Scheduling (ICAPS), pages 144–151, Providence, Rhode Island, USA.

[Gerevini et al., 2009] Gerevini, A., Haslum, P., Long, D., Saetti, A., and Dimopoulos, Y. (2009). Deterministic planning in the fifth international planning competition: PDDL3 and experimental evaluation of the planners. *Artificial Intelligence*, 173(5-6):619–668.

[Ghallab et al., 2004] Ghallab, M., Nau, D., and Traverso, P. (2004). *Automated Planning: theory and practice*. Morgan Kaufmann.

[Goldsack et al., 2009] Goldsack, P., Guijarro, J., Loughran, S., Coles, A., Farrell, A., Lain, A., Murray, P., and Toft, P. (2009). The SmartFrog configuration management framework. *ACM SIGOPS Operating Systems Review*, 43(1):16–25.

[Hagen et al., 2009] Hagen, S., Edwards, N., Wilcock, L., Kirschnick, J., and Rolia, J. (2009). One is not enough: A hybrid approach for it change planning. *Integrated Management of Systems, Services, Processes and People in IT*, pages 56–70.

[Hagen and Kemper, 2010] Hagen, S. and Kemper, A. (2010). Model-Based Planning for State-Related Changes to Infrastructure and Software as a Service Instances in Large Data Centers. In *Proceedings of the 3rd IEEE International Conference on Cloud Computing*, pages 11–18. IEEE.

[Helmert, 2004] Helmert, M. (2004). A planning heuristic based on causal graph analysis. In *Proceedings of International Conference on Automated Planning and Scheduling*, volume 16, pages 161–170.

[Helmert, 2006] Helmert, M. (2006). The fast downward planning system. *Journal of Artificial Intelligence Research*, 26(1):191–246.

[Helmert, 2009] Helmert, M. (2009). Concise finite-domain representations for PDDL planning tasks. *Artificial Intelligence*, 173(5-6):503–535.

[Helmert and Domshlak, 2009] Helmert, M. and Domshlak, C. (2009). Landmarks, critical paths and abstractions: What's the difference anyway? In *Proceedings of International Conference on Automated Planning and Scheduling*.

[Helmert and Geffner, 2008] Helmert, M. and Geffner, H. (2008). Unifying the causal graph and additive heuristics. In *Proceedings of International Conference on Automated Planning and Scheduling*, volume 8.

[Herry and Anderson, 2013] Herry, H. and Anderson, P. (2013). Planning configuration relocation on the bonfire infrastructure. In *Proceedings of the CloudCom 2013 Workshop on Using and Building Cloud Testbeds (UNICO)*. IEEE CloudCom.

[Herry et al., 2011] Herry, H., Anderson, P., and Wickler, G. (2011). Automated planning for configuration changes. In *Proceedings of the 25th Large Installation System Administration Conference (LISA '11)*. Usenix Association.

[Hewlett-Packard, 2008] Hewlett-Packard (2008). Cells as a Service. http://www.hp.com/hpinfo/newsroom/press_kits/2008/cloudresearch/fs_cellsasaservice.pdf. Accessed: 2015-02-15.

[Hewlett-Packard, 2014] Hewlett-Packard (2014). HP IDOL. http://www.autonomy.com/products/idol. Accessed: 2014-07-01.

[Hewson et al., 2012] Hewson, J. A., Anderson, P., and Gordon, A. D. (2012). A declarative approach to automated configuration. In *Proceedings of the 2012 LISA Conference*. Usenix Association.

[Hoffmann, 2003] Hoffmann, J. (2003). The Metric-FF planning system: Translating "ignoring delete lists" to numeric state variables. *Journal of Artificial Intelligence Research*, 20(20):291–341.

[Hoffmann and Brafman, 2005] Hoffmann, J. and Brafman, R. (2005). Contingent planning via heuristic forward search with implicit belief states. In *Proceedings of International Conference on Automated Planning and Scheduling*.

[Hoffmann and Nebel, 2001] Hoffmann, J. and Nebel, B. (2001). The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14(1):253–302.

[Hoffmann et al., 2004] Hoffmann, J., Porteous, J., and Sebastia, L. (2004). Ordered landmarks in planning. *Journal of Artificial Intelligent Research (JAIR)*, 22:215–278.

[Howey et al., 2004] Howey, R., Long, D., and Fox, M. (2004). Val: Automatic plan validation, continuous effects and mixed initiative planning using pddl. In *Proceedings of 16th IEEE International Conference on Tools with Artificial Intelligence*, pages 294–301. IEEE.

[HP Labs, 2014] HP Labs (2014). SmartFrog. http://www.smartfrog.org. Accessed: 2014-07-01.

[Hsu and Wah, 2008] Hsu, C. and Wah, B. (2008). The SGPlan planning system in IPC-6. *Sixth International Planning Competition, Sydney, Australia (Sepember 2008)*.

[Hsu et al., 2006] Hsu, C., Wah, B., Huang, R., and Chen, Y. (2006). New features in SGPlan for handling preferences and constraints in PDDL3.0. In *In Proceedings of 16th International Conference on Automated Planning and Scheduling*.

[IBM Corp., 2014] IBM Corp. (2014). Integrated Service Management software, IBM Tivoli. http://www.ibm.com/software/tivoli. Accessed: 2014-07-01.

[Kavoussanakis et al., 2013] Kavoussanakis, K., Hume, A., Martrat, J., Ragusa, C., Gienger, M., Campowsky, K., Van Seghbroeck, G., Vazquez, C., Velayos, C., Gittler, F., Inglesant, P., Carella, G., Engen, V., Giertych, M., Landi, G., and Margery,

D. (2013). Bonfire: The clouds and services testbed. In *Proceedings of IEEE 5th International Conference on Cloud Computing Technology and Science (CloudCom)*, volume 2, pages 321–326.

[Keller et al., 2004] Keller, A., Hellerstein, J., Wolf, J., Wu, K., and Krishnan, V. (2004). The CHAMPS system: Change management with planning and scheduling. In *Network Operations and Management Symposium, 2004. NOMS 2004. IEEE/IFIP*, volume 1, pages 395–408.

[Kephart and Chess, 2003] Kephart, J. and Chess, D. (2003). The vision of autonomic computing. *Computer*, 36(1):41–50.

[Koehler, 1998] Koehler, J. (1998). Solving complex planning tasks through extraction of subproblems. In *Proceedings of the 4th International Conference on Artificial Intelligence Planning Systems (AIPS-98)*, pages 62–69. AAAI Press.

[Lascu, 2014] Lascu, T. A. (2014). *Automatic deployment of applications in the cloud*. PhD thesis, Universita di Bologna.

[Levanti and Ranganathan, 2009] Levanti, K. and Ranganathan, A. (2009). Planning-based configuration and management of distributed systems. In *Proceedings of IFIP/IEEE International Symposium on Integrated Network Management*, pages 65–72. IEEE.

[Loughran and Toft, 2008] Loughran, S. and Toft, P. (2008). Configuration Description, Deployment and Lifecycle Management Working Group (CDDLM-WG) Final Report.

[McDermott et al., 1998] McDermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., Weld, D., and Wilkins, D. (1998). PDDL-the planning domain definition language.

[Microsoft Corp., 2014] Microsoft Corp. (2014). Microsoft System Center. http://www.microsoft.com/en-us/server-cloud/system-center. Accessed: 2014-07-01.

[Muise et al., 2011] Muise, C., McIlraith, S. A., and Beck, J. C. (2011). Monitoring the execution of partial-order plans via regression. In *Twenty-Second International Joint Conference on Artificial Intelligence (IJCAI-11)*, pages 1975–1982.

[Opscode Inc., 2014] Opscode Inc. (2014). Chef. http://www.opscode.com/chef. Accessed: 2014-07-01.

[Porteous et al., 2001] Porteous, J., Sebastia, L., and Hoffmann, J. (2001). On the extraction, ordering, and usage of landmarks in planning. In *Proceedings of the 6th European Conference on Planning*.

[Puppet Labs, 2014] Puppet Labs (2014). Puppet. http://www.puppetlabs.com/puppet. Accessed: 2014-07-01.

[RedHat Inc., 2014] RedHat Inc. (2014). http://fedoraproject.org/wiki/Anaconda/Kickstart. Accessed: 2014-07-01.

[Reynolds, 1998] Reynolds, J. C. (1998). *Theories of Programming Languages*. The Press Syndicate of the University of Cambridge.

[Richter et al., 2008] Richter, S., Helmert, M., and Westphal, M. (2008). Landmarks revisited. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence (AAAI-2008*. AAAI Press.

[Richter and Westphal, 2010] Richter, S. and Westphal, M. (2010). The LAMA planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research*, 39(1):127–177.

[Robertson, 2005] Robertson, D. (2005). A lightweight coordination calculus for agent systems. In Leite, J., Omicini, A., Torroni, P., and Yolum, p., editors, *Declarative Agent Languages and Technologies II*, volume 3476 of *Lecture Notes in Computer Science*, pages 183–197. Springer Berlin Heidelberg.

[Russell and Norvig, 2009] Russell, S. and Norvig, P. (2009). *Artificial intelligence: a modern approach*. Prentice hall.

[Schaefer, 2006] Schaefer, S. (2006). Configuration description, deployment and lifecycle management – component model. Draft 2006-03-26.

[Schmidt, 1997] Schmidt, D. A. (1997). *Denotational Semantics: A Methodology for Language Development*.

[Shah et al., 2007] Shah, J. A., Stedl, J., Williams, B. C., and Robertson, P. (2007). A fast incremental algorithm for maintaining dispatchability of partially controllable plans. In *Proceedings of International Conference on Automated Planning and Scheduling*, pages 296–303.

[Vaquero et al., 2008] Vaquero, L. M., Rodero-Merino, L., Caceres, J., and Lindner, M. (2008). A break in the clouds: towards a cloud definition. *ACM SIGCOMM Computer Communication Review*, 39(1):50–55.

[Veloso et al., 1990] Veloso, M. M., Perez, M. A., and Carbonell, J. G. (1990). Nonlinear planning with parallel resource allocation. In *In Proceedings of the DARPA Workshop on Innovative Approaches to Planning, Scheduling, and Control*, pages 207–212. Morgan Kaufmann.