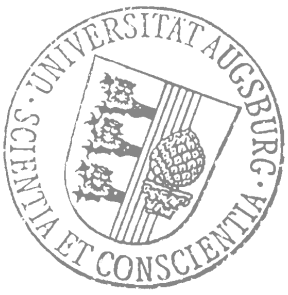





Institut für Informatik
Lehr- und Forschungseinheit für
Programmierung und Softwaretechnik
Oettingenstraße 67 D-80538 München

# Modelling and Analysing Changes in Application Landscapes: A Language and Analysis Framework for Application Landscape Management

Andrew Nathan Parker

Masterarbeit im Elitestudiengang Software Engineering

| | |
|---|---|
| Matrikelnummer: | 983270 |
| Beginn der Arbeit: | 1. März, 2008 |
| Abgabe der Arbeit: | 18. August, 2008 |
| Erstgutachter: | Prof. Dr. Martin Wirsing |
| Zweitgutachter: | Prof. Dr. Bernhard Bauer |
| Betreuer: | Andreas Schoeder (LMU), Einar Lück (IBM) |

## ERKLÄRUNG

Hiermit versichere ich, dass ich diese Masterarbeit selbständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Augsburg, den 18. August, 2008 Andrew Nathan Parker

## ABSTRACT

Datacenters used to power today's software applications are growing beyond the capabilities of the people tasked with maintaining them. Many tools have been created over the years to combat this problem, but most have focused on working with a single configuration. Since change is a constant in these environments, new tools and techniques need to be developed to help administrators work with changes to the configurations. This dissertation presents a tool to help administrators define and analyze configuration changes.

First, the current state of configuration management and planning tools is surveyed, and a gap in the current offerings is identified. A model of configurations is then presented. The model allows specification of the behavior the parts of a configuration and specification of policies that the configuration must fulfill. Next, common changes that are made to configurations are explained and used to create a list of basic configuration change operations. These operations and the model are then expanded into a programming language, called Coeus, that can be used for creating change plans. An interpreter for the language is then described, and an example of using the interpreter and language for change plan analysis is presented.

# TABLE OF CONTENTS

## ACKNOWLEDGMENTS

As I worked on this dissertation I was very surprised by the number of people who provided input, became involved, and helped me through it. A great deal of thanks must be given to my advisor at IBM, Einar Lück, for helping me through understanding application landscapes and the work being done at IBM on the topic, providing encouragement and feedback on my many half-baked ideas, and by keeping me going through his own enthusiasm for the subject. Just as many thanks have to go to my advisor at the LMU, Andreas Schroeder, for helping me stay focused and not go off on tangents, and giving me pointers to lead me through some of the more formal side of things.

Of course none of this would have been possible without the teachings of all of the professors of the Elite Studies in Software Engineering. They taught me the value of modelling techniques and opened my eyes to entire areas of software engineering that I never knew existed. Particular thanks go to Professor Martin Wirsing for acting as my mentor and helping me through my studies.

I would like to thank Narayan Desai for helping me understand what my thesis project actually was in terms that are common in the field. It was far more than one could ask for from someone who is contacted by a stranger, out of the blue, via email. I am equally thankful to all of the people at IBM who I talked to who also helped me understand what my work dealt with.

Finally, the largest thanks have to go to my wife. Without her support, encouragement, and love I do not that think moving half-way around the world, living in a place that neither of us had ever seen, nor studying for a degree in another language would have been possible.

# Chapter 1

# INTRODUCTION

> "Something went wrong," said the Space Wanderer. He sounded apologetic, as though the series of misfortunes were somehow his own fault. "A lot of things went wrong."
>
> "Have you ever considered the possibility," said Rumfoord, "that everything went absolutely right?"
>
> — Kurt Vonnegut "The Sirens of Titan"

Modeling and validating configuration changes is a necessary next step in improving system configuration management. System configuration management is a task often given to system administrators (also known as operators), who use their expertise and detailed knowledge about the computer systems under their control to make decisions. However, as these systems have grown in complexity the difficulty of reaching the correct decision has become greater and greater [32], but at the same time there are no tools known to the author that can assist system administrators in validating decisions about changes to configurations.

According to Paul Anderson the central task of system configuration management "is in determining a suitable configuration for each service on each host that will make the overall system behave according to the specification" [6]. The problem lies in the suitability of a configuration to a specification. While this seems to be a surmountable task, it has not yet received much attention from researchers or practitioners. David Oppenheimer et al. point out in their oft-cited survey of internet service failures that, although operator error is the most common cause of service failures, "operator error is almost completely overlooked in designing high-dependability systems and the tools used to monitor and control them" [32].

Oppenheimer found that operator errors occurred most often when normal maintenance was being performed on the system. It was also found that operator error was not necessarily more common than other kinds of problems, but that it was much more dangerous: more service failures were caused by operator error than any other problem. The commonalities in the failures caused by operator error were found to lie in system configuration errors. Specifically, he found that "the most troublesome configuration issues arise in specifying how components are to interact with one another" [31].

These complex, highly interconnected systems, which often contain many separate applications, have been termed application landscapes or software cities. "Like real cities, software cities need city planning to provide high quality support for the enterprise's business processes and flexible adaptability to changes in the enterprise's ecosystem" [22]. To effectively plan these application landscapes, new tools for configuration management are needed.

### *1.1 Configuration Management*

The Information Technology Infrastructure Library (ITIL) is a management framework created by the United Kingdom Office of Government Commerce to provide guidance in effectively managing large IT organizations [37]. ITIL spans most aspects of IT, but of particular interest for this dissertation are its recommendations and procedures for change and configuration management. In the context of ITIL "the goal of configuration management is to maintain a comprehensive and accurate logical representation of the IT environment" [37]. The logical representation of the environment is stored in a *configuration management database* (CMDB), which is composed of *configuration items* (CI) and their relationships.

The configurations (and configuration items) stored in the CMDB are often separated into the actual configuration and the authorized configuration. The actual configuration represents the configuration as it is expected to be found on the real machines. The authorized configurations represent the configurations that are allowed to be transitioned to.

Creation of the authorized configurations is handled by the change management procedures of ITIL. The best practice implementation of these procedures for dealing with changes is outlined in [37] as:

1. Receive a request for change (RFC)
2. Assess the change
3. Approve and schedule the change
4. Coordinate the change implementation
5. Prepare, distribute, and implement the change
6. Review the change and close the RFC

Most of the focus in creating a reliable and faithful execution of this process has been on the change implementation step, but some effort has also been placed in software that simply tracks the RFC and change through the process. An example of an implementation of the CMDB and ITIL best practices is provided by the IBM Tivoli Change and Configuration Management Database. The CCMDB provides tools for populating the CMDB, controlling changes, visualizing the CIs, and many other useful tools [28].

In order to ensure that the information stored in the CMDB is up to date, quite often these ITIL software packages include a discovery mechanism that can find the configuration of the devices in the system as well as their relationships to one another [30, 28]. They also provide automation in several other areas such as change implementation. However, the automation support in change assessment seems to be limited to visualization of the configuration.

### *1.2 Approaches to the Configuration Problem*

In addition to the processes recommended by ITIL, there have been many tools developed to deal with the configuration problem. In many cases the tools are created by individual system administrators working in a company to solve an immediate problem. This has lead to a nearly constant re-invention of the wheel, slow conceptual advancement, and low adoption levels of the tools [8]. This section summarizes some of the tools and concepts related to configuration management that have been developed over the years.

A large conceptual shift in specifying configurations was the shift from imperative (the steps to perform to reach the wanted state) to declarative (a description of the

wanted state) descriptions of configurations. The declarative method was proposed and promoted by Paul Anderson in [5] and implemented in his configuration tool LCFG. Declarative configurations have the advantage of providing what the correct configuration *is* as opposed to *how to get* the correct configuration, which allows tools to cope better with unexpected variations. Many configuration tools now use this method for specifying configurations, but it has not been universally adopted.

A different approach to combating the complexity of describing configurations is through the use of visual modelling of the configuration. UML deployment diagrams are a well known and widely used approach. Deployment diagrams can be used to convey information about a configuration such as what is installed where, the relationships between the parts, and communication paths used [19]. Another approach is presented by Andreas Hess et al. in the form of application landscapes [22]. A similar approach is described in [27] in the form of software cartography. Both approaches provide a process by which the components of the architecture of an enterprise can be identified, relationships found, and the whole system visualized. These visualization techniques provide an easier way for humans to consume the information but do not provide a fundamental change in how configurations are understood.

In addition to ways of describing configurations, a number of tools have been created and made available to assist system administrators in applying configurations to the components of a computer system [17]. Bcfg2 and LCFG are two good examples of common tools that provide a way of defining the configuration in a high-level language by way of key/value pairs that are interpreted by various subsystems to provide a concrete configuration, which is then applied to the computers in the system. These two tools do not, for the most part, provide a way for checking that the given configuration is correct. The Quattor system uses the Pan language which can define constraints on parts of the configuration [13], thereby allowing more precise and reliable descriptions.

The IBM Dynamic Infrastructure (IDI) is a tool for managing datacenters in a more service oriented fashion [24]. It approaches configurations by providing offerings of a service. Users subscribe to the offerings at which point resources are devoted to fulfilling the subscription. IDI deals with changes by issuing orders. An order describes the change that needs to be made to the resources that are a part of the subscription. IDI is very focused on changes to the configuration but has a very weak model of the configuration

itself.

SmartFrog from Hewlett-Packard is a framework to enable distributed and autonomic configuration management. The SmartFrog framework provides a component model to "support the various lifecycle operations such as creation, versioning and termination, as well as management actions such as accessing status information" [23, pg. 10]. Additionally, SmartFrog includes a configuration description language that allows for validation of configurations and a configuration management system to manipulate and deploy these configuration descriptions. Since SmartFrog itself does not have the ability to apply the configuration to a computer, it needs to be combined with other tools to be useful for managing systems. An example of this being done is provided in [7] where SmartFrog was used for dynamic reconfiguration and LCFG was used to apply the configurations.

PoDIM takes a slightly different approach for modelling, changing, and checking compliance of configurations [16]. PoDIM provides a way of modelling and checking cross machine constraints as well as the ability to modify the configuration from within the PoDIM system. The PoDIM language uses a declarative style to constrain the set of possible configurations and then uses model finding to create an actual configuration that can then be applied with a configuration engine (such as LCFG).

Recent conceptual advancement in the field has been made by Mark Burgess and Alva Couch. Mark Burgess researched and proposed a new way of modelling configurations using what is called promise theory [12]. Promise theory provides a unified way of describing many of the various aspects of configurations that have been used in many other approaches and a method for modelling communication in unreliable networks [11]. The core idea is that agents in a configuration *promise* to provide certain services to other agents. The promising agent, however, is not bound to actually provide the service. Promises are designed to be used to describe interactions within loosely coupled distributed systems. Alva Couch, on the other hand, proposed his idea of closures as a way of creating predictability from unpredictable components by packaging them in a way to be able to guarantee certain behaviors [15]. The closure presents conduits (interfaces) through which the component can be manipulated. At the same time the closure is bound by policies and will guarantee certain behavior as long as the conduits are used.

### *1.3 Filling the Gap in Configuration Management*

Unfortunately, none of these tools or techniques cover the assessment step of the ITIL process very well. For assessment one needs to examine the current configuration and decide on the changes that need to be performed. Once those changes are determined then the individual steps can be scheduled. Each individual step can be dealt with using many of the tools described above. To deal with executing each step at the correct time, the authors of Bcfg2 created an extension that can execute a series of changes in the correct order [18]. What is still missing, however, is making sure that the *correct plan* is executed; currently, most work deals with executing the *plan correctly.* Since "change occurs frequently in most environments" [18], executing the correct plan is just as, if not more, important as executing the plan correctly.

None of the tools described above provide a way for the system administrators to automate reasoning about changes to the configuration. The fundamental deficiency that these tools have in solving the problem faced by system administrators is summed up by Alva Couch when he argues that these tools simply raise the level of abstraction; they "do not 'make things easier to understand'; they make things that remain difficult to understand easier to construct" [14]. SmartFrog and PoDIM appear to be the most promising tools available for aiding change analysis since they can remove much of the burden from the administrator. Unfortunately, neither is designed for analysis. SmartFrog is meant to be used for autonomic configuration management, and PoDIM is designed as a better way to deal with centrally controlling configuration definition and deployment.

The visual approaches provide a means that has proven to be useful for understanding large systems. Although all three styles provide a useful way of visualizing the current configuration of a system, they provide very little in the way of either determining the suitability of, or reasoning about changes to the configuration. The administrators must still work through all possible impacts themselves and understand the behavior of each component of the system.

Promise theory does provide a way for reasoning about autonomous configuration agents that do not need intervention from system administrators to overcome obstacles. Closure theory provides the structure for creating predictability from unpredictable

components and a framework for discussing them. However, once again neither theory directly solves the problem that is faced by system administrators when they need to plan multi-step changes to a system.

What is needed is a tool that can fill the analysis gap in the ITIL change management process. Each of the tools provides a piece of the puzzle, but the correct pieces need to be selected and placed together. The ideas of relationships from the CMDB and promises from the promise theory give a way of working with the inter-dependencies in configurations. Closure theory provides a structure for describing components that have behavior and policies limiting them. Finally, subscriptions from IDI provide a way of splitting up configurations to deal with different parts having to conform to different policies.

### *1.4 Structure of this Dissertation*

This dissertation presents a tool to assist system administrators in planning changes to a configuration. The tool is a first step in filling the gap identified above and pointed out by David Oppenheimer when he stated that "the complexity of component relationships and configuration options suggests that radically new tools are needed for tasks such as visualizing current and past configurations, and predicting the global impact of changes in component configurations" [31].

The next chapter defines a model for configurations that includes describing the behavior of the configuration components and policies that the configuration must conform to. The chapter 3 investigates changes that are made to configurations in order to identify the operations of a "change language" that can be used for describing a sequence of configuration changes.

Chapter 4 ties the previous two chapters together by defining a complete language and an interpreter for the language. The interpreter allows for analysing changes to determine if they conform to the policies that are in place for the configuration. An example of using the tool for analysing a change is also given. Finally, chapter 5 concludes the work and presents several ideas for future directions of implementation and research work.

Chapter 2

# A MODEL OF SYSTEM CONFIGURATION

There are many possibilities when modelling system configurations. One can model the filesystem layouts of the systems in a way that the model contains where and in what form configuration elements are stored, or the model can contain the logical structure of the systems by having parts of the whole connected together in their logical relationships. Models of individual configuration elements could be designed to model the complex, heterogeneous, almost unstructured structures that appear in the configuration files of most programs, or they could be designed to allow reuse and strict type checking of configuration elements to make them easier for a human to understand and limit errors. The list of possible model variations is vast, but none fully suited to analyzing the impact of changes.

The configuration models used by most configuration tools do not hold information about the dependencies between parts in a system. This means that any kind of reasoning about indirect effects of changes are extremely difficult, if not impossible in those models. A model that concentrates on relationships between parts as well as the behavior of the parts themselves is needed. Without the information about dependencies any change to the system could result in broken dependencies. Broken dependencies may show up immediately or may lurk in the background only to appear at the most inopportune moment, wreaking havoc as data is lost and customers' wishes are not served ([32] relates several such stories).

The work done in [11], [15], [24], and [26] suggests another way of modelling configurations. Instead of looking at simple key-value pairs in a mostly unstructured fashion, constraints can be placed on the configuration as a whole and relationships between the parts can be modelled. This chapter deals with creating a model of configurations to support describing and analysing configuration change plans.

First, a way of describing the parts, as well as the whole, of configurations is presented. The description of configurations is then enriched with the ability to express constraints on configurations. Finally, a way of combining configurations together is

given, which allows for reasoning about the consequences of changes across parts of systems with differing constraints.

### *2.1 Instances, Usages, and Configurations*

A configuration, at the core, remains a simple mapping from names to values. Every part of a system (computer, disk, daemon, etc.) has a set of parameters that control its behavior. Each of these parts is of a particular "type", of which each is an instance. The type of an instance determines how the parameters are interpreted to produce the instance's behavior. However, instances are not always in an active state. At times they are stopped and do not have any behavior. Therefore, instances also have a running state which describes whether the instance is operating or not[1]. These concepts can be formalized as:

$$\begin{aligned} \mathit{Behavior} &= \mathit{Inst} \times \mathit{Conf} \rightarrow \mathcal{P}(\mathit{Label}) \\ \mathit{Params} &= \mathit{Label} \rightarrow \mathit{Value} \\ \mathit{RunState} &= \{\text{running}, \text{stopped}\} \\ \mathit{FailState} &= \{\text{failed}, \text{working}\} \\ \mathit{Inst} &= \mathit{Params} \times \mathit{RunState} \times \mathit{FailState} \\ \mathit{type} &: \mathit{Inst} \rightarrow \mathit{Behavior} \end{aligned}$$

As can be seen, an instance $i \in \mathit{Inst}$ consists of a mapping for its *Params* and its current *RunState*. The *FailState* tracks if the instance has failed and will be discussed later when policies are presented. The type of $i$ is determined by the function *type*, which provides the $b \in \mathit{Behavior}$. The behavior of an instance then maps an instance as well as the configuration $c \in \mathit{Conf}$, in which the instance can be found, to a set of *Label*s. Each of the *Label*s represents a capability of the instance. These capabilities can be used by the instance itself as well as by other instances in order to drive more behavior.

The configuration in which an instance finds itself can be represented as an edge-labeled, directed graph of instances, where the instances are the nodes of the graph and dependencies between instances are the edges. There are many kinds of dependencies possible, such as version dependencies (as expressed in packaging systems such as RPM) or lifecycle dependencies (a piece of software can only run if the computer on which it

[1]There are more states that could be modelled as seen in the SmartFrog system [23, pg. 106]. However, the extra states do not add interesting information for modelling the system in this case.

is installed is turned on). The dependencies modelled here will be limited to dependencies of one instance using a service of another instance (usage relationships) and an instance being hosted on another instance (hosting relationships, a special case of usage relationships, and a form of lifecycle dependencies).

$$\begin{aligned} \mathit{Usage} &= \mathit{Inst} \times (\mathit{Label} \cup \{\text{hosting}\}) \times \mathit{RunState} \times \mathit{FailState} \times \mathit{Inst} \\ \mathit{Conf} &= \mathcal{P}(\mathit{Inst}) \times \mathcal{P}(\mathit{Usage}) \end{aligned}$$

In addition to instances having a *RunState*, the usage dependencies have one as well. This provides a way of modelling suspended relationships in the system (such as backup database connections). The *Usage*s also have *Label*s that identify which capability one instance is using from the other. However, in addition to the standard labels, there is a special label for *Usage*s to mark hosting relationships. These are the relationships that identify lifecycle dependencies (such as a piece of software installed on a piece of hardware).

In working with these structures a notation is needed to reference the various parts. It is assumed that there is a collection of projections $X_Y : X \to Y$ for each of the structures defined above where $X$ is the structure and $Y$ is the part of the structure. For example, $\mathit{Inst}_{\mathit{Params}}(i)$ provides the *Params* function of instance $i$. In the case of *Usage*s where there are two *Inst* positions, they will be named to match the meaning of the position in the relationship so that *user* refers to the leftmost *Inst* position and *provider* refers to the rightmost.

Not all possible configurations will be allowed. Only well-formed configurations will be considered. Well-formed configurations fit the following rules.

**Well-Formed Graph** For a given configuration $c$, the endpoints of edges (*Usage*s) must be nodes (*Inst*s) in $c$.

$$\forall (u, l, r, f, p) \in \mathit{Conf}_{\mathit{Usage}}(c) : u \in \mathit{Conf}_{\mathit{Inst}}(c) \wedge p \in \mathit{Conf}_{\mathit{Inst}}(c)$$

**No Ghosts** All well-formed behaviors return the empty set when the provided instance has a *RunState* of stopped.

$$\forall i \in \mathit{Inst}, b \in \mathit{Behavior}, c \in \mathit{Conf} : \mathit{Inst}_{\mathit{RunState}}(i) = \text{stopped} \Rightarrow b(i, c) = \emptyset$$

**No Floating Instances** When an instance has a *RunState* of stopped all other instances hosted on it, within the configuration $c$, are also stopped.

$$\forall (u, l, r, f, p) \in \mathit{Conf}_{\mathit{Usage}}(c) : l = \text{hosting} \\ \wedge \mathit{Inst}_{\mathit{RunState}}(p) = \text{stopped} \Rightarrow \mathit{Inst}_{\mathit{RunState}}(u) = \text{stopped}$$

**No Multi-hosting** An instance can only be the *user* in one hosting relationship within a configuration $c$.

$$\forall i \in \mathit{Conf}_{\mathit{Inst}}(c) : \\ \left|\left\{u \in \mathit{Conf}_{\mathit{Usage}}(c) \mid \mathit{user}(u) = i \wedge \mathit{Usage}_{\mathit{Label}}(u) = \text{hosting}\right\}\right| \leq 1$$

**No Impossible Uses** The label of a usage relationship, which is not stopped, between two instances $u$ and $p$ must be in the set of behaviors of $p$ within a configuration $c$.

$$\forall (u, l, r, f, p) \in \mathit{Conf}_{\mathit{Usage}}(c) : l \neq \text{stopped} \wedge l \neq \text{hosting} \Rightarrow l \in \mathit{type}(p)(p, c)$$

### *2.2 Landscapes, Policies, and Subscriptions*

In addition to the configurations that were described in the previous sections, quite often it is useful to talk about subsets of the configuration and require additional invariants on these sub-configurations. By building upon the concepts from [15], [22], and [24] another level of encapsulation and specification can be added to configurations.

Policies provide additional structure for a configuration. A policy is nothing more than a predicate used to determine if a given configuration conforms to some set of requirements. Policies can be used to ensure that configurations provide certain services, have a required amount of redundancy, or do not enter a state that is known (from experience) to be incorrect.

Having one policy over all time is not a reasonable representation of how configuration management of large computer systems works. Different policies need to be in effect depending on what needs to be done. Quite often there is, for example, a distinction made between when a system is running normally and when maintenance needs to be performed. In the former case there may be much more stringent requirements for availability and capacity than in the latter. This means that there are not only

policies that describe the set of allowed configurations but also landscapes that define the collections of policies that will be allowed for various sub-configurations. This concept of policies plus landscape corresponds closely to the set of all reasonable policies defined by Couch: "A policy is not a definition of what must happen, but rather a list of reasonable options" [15].

These concepts of policies and landscapes can be formalized as follows.

$$\begin{aligned} \mathit{Policy} &= \mathit{Conf} \to \{\mathit{tt}, \mathit{ff}\} \\ \mathit{Land} &= \mathcal{P}(\mathit{Policy}) \end{aligned}$$

This formulation of policies, although useful for talking about and reasoning with this model, does not help in understanding how a policy is defined. Policies are open to any kind of expression that has the ability to make a decision about a configuration, but, for this work, the language of policies has been limited to boolean expressions in the Coeus language. In addition to simple boolean expressions a policy can also check the configuration for resiliency. Resiliency is the ability of a configuration to conform to the policy even after a given number of *Inst*s or *Usage*s have failed, which is the reason for the *FailState* component of instances and usages. This allows defining that configurations may not have single points of failure, for example. A more exact explanation of the structure of policies, how resiliency is defined, and how the configurations are checked against the policies can be found in chapter 4.

The combination of a configuration $c \in \mathit{Conf}$ and a landscape $l \in \mathit{Land}$ is a subscription. The subscription represents the need for certain policies to be used in order to provide a service. Since a *Land* can have many policies, the state of a subscription is also needed to track which policy is currently in effect for its configuration.

$$\begin{aligned} \mathit{Sub} &= \mathit{Conf} \times \mathit{Land} \\ \mathit{state} &: \mathit{Sub} \to \mathit{Policy} \end{aligned}$$

The *state* mapping has the reasonable constraint that it must always provide a policy that is part of the landscape for the subscription. Deciding if a subscription $s$ conforms to its *Land* becomes quite easy to formulate:

$$\mathit{conform}(s) \iff \mathit{state}(s)(\mathit{Sub}_{\mathit{Conf}}(s)) = \mathit{tt}$$

### *2.3 Systems of Configurations*

To allow for multiple subscriptions, each with its own policies and landscape, and to analyze them as a whole, requires a structure to tie the parts together. If policies are to apply to sections of configurations, then it implies a sort of hierarchy of configurations. Such a hierarchy can be achieved by introducing a "system"[2] that encompasses all configurations and subscriptions that are in effect at a given time. This system could be understood to be a single datacenter or all computers on the planet, and the subscriptions are the various configurations that have been created to suit the needs of various users. Having the complete system which contains the subscriptions provides a way of reasoning about instances that are shared between subscriptions as well as those instances that are not part of any subscriptions (such as computers that are part of a pool). The structure of a system is:

$$Sys = \mathcal{P}(Sub) \times Conf$$

In order to ensure that a system $m$ truly does represent the entirety of instances and usages it must contain all instances and usages from all of the configurations within its subscriptions. The usage relationships that exist in the system-wide configuration also require that certain instances and usages need to be existent in the subscription configurations. Therefore, for a system to be well-formed, all subscription configurations must be in a particular relation to the system configuration.

$$\forall s \in Sys_{Sub}(m) : Sub_{Conf}(s) \sqsubseteq Sys_{Conf}(m)$$

Where $\sqsubseteq$ for two configurations $c_1$ and $c_2$ is defined as:

$$\begin{aligned} c_1 \sqsubseteq c_2 \iff & \quad Conf_{Inst}(c_1) \subseteq Conf_{Inst}(c_2) \\ \wedge & \quad Conf_{Usage}(c_1) \subseteq Conf_{Usage}(c_2) \\ \wedge & \quad \forall u \in Conf_{Usage}(c_2) : \\ & \quad user(u) \in Conf_{Inst}(c_1) \\ & \qquad \Rightarrow (u \in Conf_{Usage}(c_1) \wedge provider(u) \in Conf_{Inst}(c_1)) \end{aligned}$$

This means that all of the subscription's instances and all of its usages must also be in the system and that anything that the instances of the subscription use must also be

[2]The term *system* has become so misused in the IT field that its use is now quite often more confusing than enlightening. For this dissertation I have tried to limit its use to the meaning of system that is given here.

in the subscription. This definition has the advantage of making subscriptions closed systems, where everything that they are using is considered a part of the subscription. There is, however, also a disadvantage whereby it becomes harder to model external services that a subscription may be using (such as an e-commerce payment provider). This limitation can be circumvented by modelling interfaces to the external services as instances and thereby creating a separation between the external service itself and the interface to it.

Now that instances can be in many configurations at once the question arises as to which configuration is used when calculating the capabilities of an instance. Previously the signature of a behavior function $b$ for instances was given as $b : \mathit{Inst} \times \mathit{Conf} \rightarrow \mathcal{P}(\mathit{Label})$. The capabilities of an instance could be local to each individual subscription or a single configuration can be chosen that is used for deciding the capabilities of instances. The former choice has the advantage of making the behavior of each instance very clear within the context of a subscription, but does not match well with reality. The subscriptions that have been presented here are almost entirely a logical construction to ease reasoning about the configurations, and in actual configurations there is no such separation. It would be strange to assume that an instance could behave differently depending on who or why someone is observing it. Therefore, the behavior of instances will always be determined in the largest context possible: the system.

The conformance of the individual subscriptions in a system can be extended to define conformance for the system as a whole. A system $m$ is conformant when:

$$\mathit{conform}(m) \iff \forall s \in \mathit{Sys}_{\mathit{Sub}}(m) : \mathit{conform}(s)$$

Now that many subscriptions are tied together through the system, interrelationships between subscriptions can be studied. There are at least two interesting properties of subscriptions when planning changes to the configurations. The first property is that of independence of subscriptions. Two subscriptions are independent of each other when they have nothing in common. If they are not independent, then they have a high possibility of effecting one another since there is an instance or usage that they share.

$$\mathit{independent}(s_1, s_2) \iff \mathit{Conf}_{\mathit{Inst}}(\mathit{Sub}_{\mathit{Conf}}(s_1)) \cap \mathit{Conf}_{\mathit{Inst}}(\mathit{Sub}_{\mathit{Conf}}(s_2)) = \emptyset$$

Of course two subscriptions not being independent is not itself interesting during planning. The real question is if one subscription is limiting the other in some manner;

if one is imposing some "hidden" policy. A subscription is constraining another subscription if a change is made to the configuration of one subscription which causes the other subscription to no longer be conformant, meaning that the attempted change is not allowed.

In order to formulate the constraining relationship, there needs to be a way of expressing changes to systems. A change to a system can be expressed as a function that, given a system, provides a new system. The set of changes will be referred to as $\Delta$.

$$\Delta = Sys \rightarrow Sys$$

The set of allowed changes for a system $m$ is all changes whereby after applying the change the system is conformant.

$$allowed(m) = \{\delta \in \Delta \mid conform(\delta(m))\}$$

This then allows defining subscription $s_1$ to constrain subscription $s_2$ in a system $m$ to be when the set of allowed changes that affect $s_2$ to be a strict subset of those when $s_1$ is not present in the system.

$$constrains(s_1, s_2, m)$$
$$\iff \{\delta \in allowed(m) \mid s_2 \neq \delta(m)_{s_2}\} \subset \{\delta \in allowed(m \backslash s_1) \mid s_2 \neq \delta(m)_{s_2}\}$$

Where the notation $m \backslash s$ is used to denote the system $m$ with the subscription $s$ removed, and the notation $m_s$ is used to denote the projection of the subscription $s$ from the system $m$.

It needs to be noted that for two subscriptions $s_1$ and $s_2$, in a system $m$, there is no correlation between independence and constraint. Specifically, it is not true that $\neg independent(s_1, s_2) \Rightarrow constrains(s_1, s_2, m)$ nor is it true that $constrains(s_1, s_2, m) \Rightarrow \neg independent(s_1, s_2)$. This can be seen in the fact that the behavior of an instance is based upon the configuration of the entire system, which allows any change to affect any subscription.

Chapter 3

# CONFIGURATION CHANGES

The previous chapter presented a model that provides a structure in which to discuss configurations. Several details of the model were, however, left open. One of those open points was the elements of $\Delta$—the changes to the system. To discover what the elements of $\Delta$ are, real world change scenarios need to be examined. The scenarios will provide a basic list of operations that can be expanded upon to create a complete language for describing and analysing configuration changes.

After the scenarios are presented, a list of individual operations is compiled, which forms the basis of the changes to the configuration model. The next chapter expands upon these operations to provide a full language for planning and analysing changes.

## *3.1 Configuration Management Scenarios*

Each scenario is composed of five parts:

- A **name** used to identify the scenario.
- A short **summary** of the situation in which such a change would need to be made and what it does.
- The **steps** needed to carry out the change.
- A **discussion** of the change where variations and difficulties are brought up.
- An **example** in which the change is performed.

These scenarios would ideally be taken from a reference work on common changes and patterns in system configuration management. Unfortunately, it seems that there is very little work available in this area. Most work concentrates on what the elements of the configuration should be, rather than how those elements are manipulated. These

scenarios were therefore created from the author's own experiences, thoughts, and input from his advisors, and from Narayan Desai of the Bcfg2 project.

---

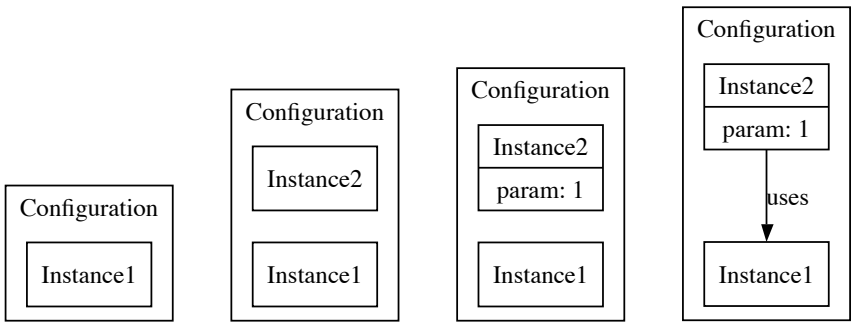
# Commission Instance

## Summary

A new instance is needed to fulfill some, possibly new, requirement. A new instance is commissioned and connected to other instances.

## Steps

The configuration begins in some initial state (Figure 3.1(a)). A new resource is added by selecting the type of resource and provisioning a new instance of it (Figure 3.1(b)). Any needed parameters are set on the instance (Figure 3.1(c)). Finally, any dependencies on other instances are created by commissioning usage relationships (Figure 3.1(d)). These last two steps together provide the settings for the instance to be able to provide the capabilities needed.




(a) Initial configuration (b) Commission instance (c) Set parameters (d) Connect instances

Figure 3.1: Commissioning a new instance

## Discussion

Resources that can be commissioned, thus creating an instance, are not limited to hardware (computers, disks, network cards, etc.), but can also be software. In most cases commissioning an instance is actually an act of installing it on some other instance (such as software installed on an operating system, or a network adapter installed in a

computer). In this case a hosting relationship needs to be created between the hoster and hostee.

When the behaviors of instances are being defined, care needs to be taken to ensure that all of the important aspects of the behavior and possible dependencies are modelled. This is needed to minimize the problem of "hidden preconditions" [15] and thereby increase the reliability of an analysis. An example of behaviors that may be easy to overlook are voting algorithms in distributed systems that use broadcasting to communicate. Such a relationship could be modelled by explicitly modelling the voting groups with usage relationships, but could also be modelled by making the discovery of other instances a part of the behavior of the instances.

### Example

The system provides a remote file system (NFS, SMB, or some other protocol for access) for the storage of documents. A SAN attached storage subsystem[1] manages the logical volumes used for storage. A fileserver provides clients access to the file system and is attached to the storage subsystem via the SAN. The currently allocated storage is no longer sufficient and more space must be added.

The first step is to create a new logical volume on the storage subsystem. The storage subsystem makes the volume available on the SAN. Finally, the fileserver begins using the new volume for storage.

---

## Decommission Instance

### Summary

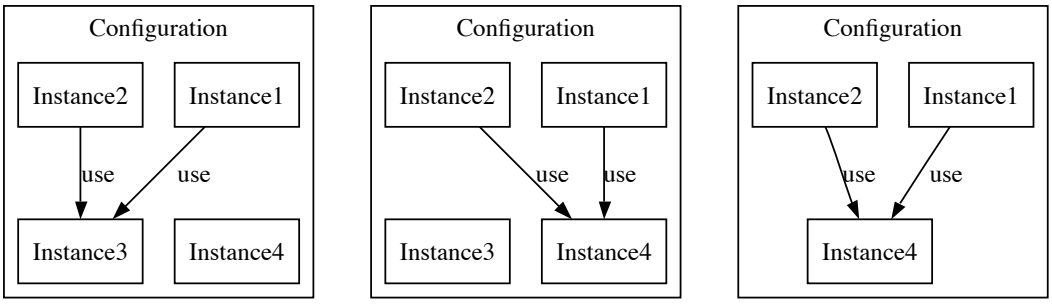
The system being managed has excess capacity, functionality, or an instance needs to be retired. First remove all dependencies on the instance and then remove the instance itself from the system.

### Steps

The first step is to identify all of the dependencies on the instance that are present in the system (Figure 3.2(a)). All dependent instances must then be updated to no longer be dependent (Figure 3.2(b)). At this point the instance can be decommissioned

[1] "A storage subsystem is a device that coordinates and controls the operation of one or more disk drives. A storage subsystem also synchronizes the operation of the drives with the operation of the system as a whole." [25]

and removed from the system (Figure 3.2(c)).




(a) Initial configuration (b) Change dependencies away (c) Decommission instance

Figure 3.2: Decommissioning an instance

## Discussion

Decommissioning an instance can be a very disruptive task because of its possible impact on other instances that depend, in some way, on the instance to be decommissioned. Removing a single instance might necessitate reconfiguring many others. Successfully decommissioning an instance depends heavily on correctly ordering the individual operations performed to keep the system as a whole working. If an instance is removed before the rest of the system is ready, then it may cause a failure of the entire system.

Depending on how other instances in the system depend on the instance to be decommissioned, different actions need to be taken. The dependent instances may need to be decommissioned as well or a suitable replacement for the instance to be decommissioned needs to be found. Usage relationships are often much easier to deal with since they simply require changing the dependency to a suitable replacement. Hosting relationships require either the hosted instances to be decommissioned as well or finding a way to migrate them to a new host.

## Example

The system is a cluster of web servers. Requests to the web servers are distributed among them by a load balancer. Additionally, the access logs created on each web server are aggregated by a log processing server. One of the web servers needs to be removed

from the cluster.

The first step is to update the load balancer to remove the web server from its load balancing pool. The log aggregation server is then updated to stop pulling logs from the web server. All dependencies on the web server have now been removed from the system. The web server can now be decommissioned.

---

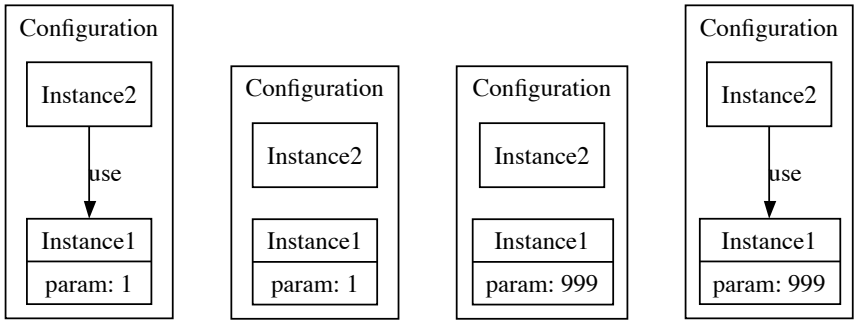
# Modify Instance Parameters

## Summary

The parameters of an instance need to be changed in order to modify the capabilities. This requires taking the instance offline, updating the parameter, and placing the instance back online.

## Steps

Identify the instance that needs to updated and any dependencies on it (Figure 3.3(a)). If the instance must be taken offline to perform the change, then all dependent instances need to be updated to not depend on it (Figure 3.3(b)). Once dependencies on the instance have been dealt with, the parameters of the instance itself can be changed to the new values (Figure 3.3(c)). Once the instance is running with the new parameters, the other instances can use the instance again (Figure 3.3(d)).




(a) Initial configuration (b) Remove dependencies (c) Modify parameters (d) Reinstate dependencies

Figure 3.3: Modifying instance parameters

## Discussion

Modifying the parameters of an instance may seem to be a straightforward change, but on closer inspection many problems and traps begin to emerge. Since the parameters of an instance determine what sorts of services the instance will be able to provide to other instances, a change of parameters can quite easily lead to changing an instance's capabilities in such a way that it is no longer compatible with usage dependencies in place. Other problems can occur because the implementation of the instance may need to shut down completely in order to take on the new parameters. This can lead to disruption of all usage dependencies for a period of time, even if there is no change in the services provided.

The steps for actually modifying the parameters can change depending on the implementation of the instance. Some implementations may need to be restarted in order to pick up the changes, others may need to be informed of the change (such as through sending the process a SIGHUP), and others may detect the change on their own. Each of these presents problems and advantages. A process that must be restarted will interrupt its service to dependents, but it is easy to determine when the change has taken effect (once the instance is running again it can be assumed that it is running with the new parameters). A process that can detect the changes without any intervention will most likely not cause a disruption in service, but it can be difficult to determine if it has yet made the change.

A type of parameter modification that may not be immediately clear is that usage dependency changes are also parameter changes. Most of the dependencies between parts of a configuration are expressed by entries in configuration files (such as the DNS servers recorded in a Unix computer's `/etc/resolv.conf` file). This means that the act of changing usage dependencies also results in the difficulties described here; therefore, care must be taken when analysing and planning changes to usage dependencies.

## Example

An Apache web server using virtual hosts to serve two different websites needs to be tuned to improve performance. The MaxChildren parameter needs to be increased in the Apache server to allow it to serve more requests at once. However, one of the virtual hosts is running in a cluster with a load balancer in front of the web server, and the other is the sole server for its site.

The first step is to modify the configuration of the Apache instance. Apache servers do not immediately take up new configurations and must be restarted for the changes to take effect. Since shutting down the server will shut off both virtual hosts, the next step is to remove the load balanced host from the load balancer, but since the virtual host for the other site does not have this option it must accept a service interruption. The Apache server is stopped and restarted. At this point the host is added back into the load balancer's pool.

---

# Patch Software

## Summary

The software currently running as an instance needs to be updated. This requires taking the instance offline, updating the software, possibly changing parameters, and placing the instance back online.

## Steps

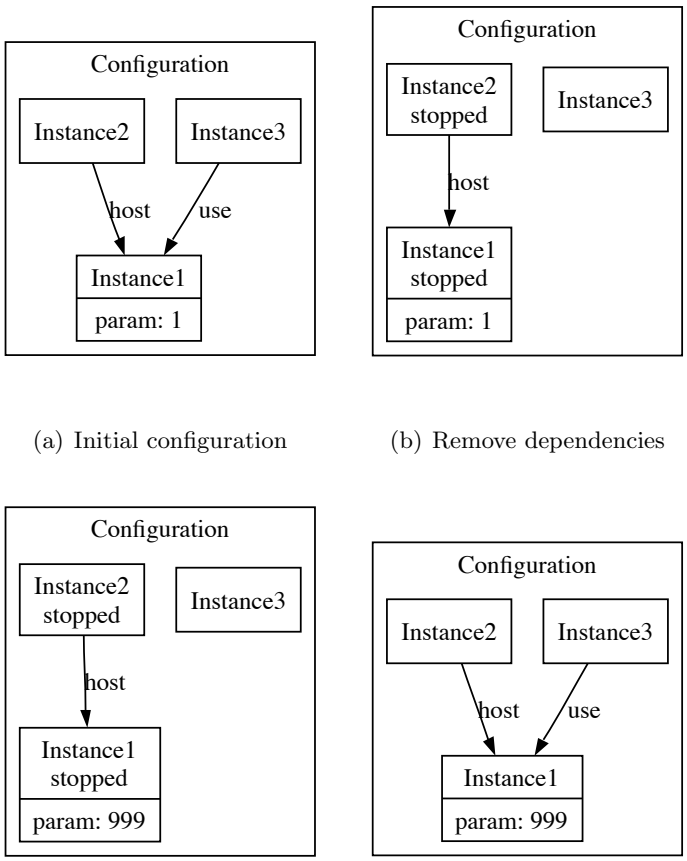
Identify the instance that needs to be updated and any dependencies (Figure 3.4(a)). Hosted instances will need to be stopped along with the instance to be updated, and other dependencies will need to be removed (Figure 3.4(b)). The instance that is to be updated is taken offline and the update is installed. Changes are made to the instance parameters to match any changes required by the update (Figure 3.4(c)). The instance is brought back online and the other instances are restored to being dependent on the updated instance once again (Figure 3.4(d)).

## Discussion

Patching software is a very risky change in system configurations, since the entire purpose of patching is usually to change the behavior of the thing being patched. Often the changes are known and anticipated, but quite often the patch has an unexpected interaction with other software in the system. For this reason it is often recommended that correctness testing of components (software and hardware) is done before putting them into a production environment as well as online testing done during their normal operation [32].

## Example

The system is a set of two web servers running with a load balancer in front of them.




(a) Initial configuration (b) Remove dependencies

(c) Modify parameters (d) Reinstate dependencies

Figure 3.4: Patching software

The web servers are connected to a relational database. A second relational database instance acts as a hot spare to the main database and uses the main database to maintain a constantly up to date copy of the data. The database software on the main database needs to be updated. First, the web servers need to be updated to use the second database instance, and the second database instance needs to stop replicating the main instance. The main database can now be taken offline, have its software updated, and be brought back online. The main database now needs to replicate the second database instance to catch up with the changes that have been performed there. Once the main database is up to date with the backup, the web servers can be changed back to use the main database instance, and the backup instance can begin replicating the main

database again.

### *3.2 Requirements for Changes*

There are only a few operations that are needed to describe the above scenarios. Most of the changes should be obvious to anyone who has used a computer: installing, uninstalling, and changing parameters of components; adding and removing dependencies; starting and stopping components. Each of these operations can be very easily mapped to the configuration model that was presented in the previous chapter. Table 3.1 lists the changes to configurations that are required by the scenarios.

Table 3.1: Configuration modifying operations

| **Operation** | **Description** |
|---|---|
| **commision** | Create a new instance. |
| **decommision** | Remove an existing instance. |
| **install** | Create a hosting relationship between two instances. |
| **set parameter** | Set a configuration parameter on an instance. |
| **use** | Create a usage dependency between two instances. |
| **unuse** | Remove a usage dependency between two instances. |
| **stop instance** | Put an instance into a state in which it cannot fulfill the usage dependencies on it. |
| **start instance** | Put an instance into a state in which it possibly can fulfill the usage dependencies on it. |

It should be noted that there is a special operation just for installing an instance on another instance. There are two reasons for specifying installation as a basic operation. Firstly, many people view commissioning a new piece of hardware and installing something as different activities, and so, by separating commissioning and installing, the operations will more closely match the way users think about changing systems. Secondly, a hosting relationship is special in that it means that there is a lifecycle dependency from one instance to another, and so the creation of the relationship should be treated differently from other relationships.

In addition to operations that modify the configuration, there are also operations that do not modify the configuration, but rather provide information about what is already there. These are needed to discover the instances and relationships in a configuration that need to be modified. As there are only two types of objects in a configuration, only two types of querying operations are needed: finding dependencies and finding instances. However, what kinds of queries need to be supported so that the necessary dependencies and instances can be found?

Instances need to be able to be found by type (find all web servers), parameter values (find all instances using port 80), lifecycle status (find all instances that are running), capabilities (find all instances that have capability "foo"), and any combination thereof. Since the decision was made to treat hosting relationships specially when creating them, they should also be treated specially for queries, as well. Therefore, there needs to be a way to find instances based upon hosting relationships.

Usages are much simpler objects than instances and so querying for them is also much simpler. Since usage relationships are to use a capability provided by an instance they can be queried for in those terms: find usages by user $x$ for capability $y$ provided by $z$. There also needs to be a form for finding all usages regardless of the capability being used to support finding all dependencies on some instance.

The Patch Software scenario presents an interesting question: patching software is a very important, and common, activity in system administration, but is it something that is needed for describing configuration changes? When planning and analysing configuration changes, the important question is if the planned change will result in a system that no longer fulfills its requirements. When patching software (or installing new firmware, or any one of the various tasks that are similar) the important aspect for planning or analysing is not the act of patching itself. Instead, the important aspect is that one set of behaviors is leaving the system and a new set of behaviors is entering it. This suggests that patching does not need to be an operation and can instead be understood as decommissioning one instance and commissioning a new one to replace it that exhibits the new behavior. If the planned patch does not involve significant change to the behavior of an instance, then the planning of patching can be simplified even further.

Chapter 4

# THE COEUS LANGUAGE: CONFIGURATION EVALUATION SYSTEM

> **Prometheous:** Often my mother Themis, or Earth (though one form, she had many names), had foretold to me the way in which the future was fated to come to pass. That it was not by brute strength nor through violence, but by guile that those who should gain the upper hand were destined to prevail.
>
> — Aeschylus "Prometheous Bound"

Before diving into the language itself, let us revisit the workflow for managing configuration changes in order to recall how all of what has been presented so far fits into change planning. The basic workflow for change planning and management begins with a change request. An impact analysis is performed based upon the current configuration of the system, and a plan of configuration management tasks is created. Once the analysis has been reviewed and the tasks approved, the plan is scheduled and then executed [29, pg. 67-69]. Currently, the impact analysis is done by technical experts and relies heavily on their reasoning abilities.

Instead of relying solely on an expert's reasoning, the configuration can be modelled, the tasks described in terms of the change operations, the policies defined, and the plan checked. The current system configuration[1] must first be translated to the model from chapter 2. Since the actual configuration does not divide instances in the system by subscription or contain policy and landscape information, the expert next defines the policies in the system, creates subscriptions, and assigns the instances to the appropriate subscriptions. The sequence of changes that needs to be performed to fulfill the change request is described in terms of the change operations that have been identified in the

[1]Remember that system configuration (the *Conf* of the *Sys*) refers to the entirety of the computers, software, and whatever else that is being considered (see section 2.3).

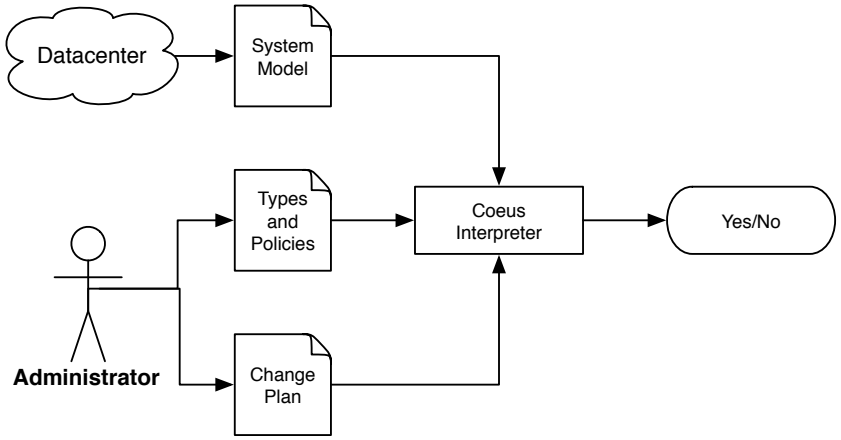



Figure 4.1: Administrator creates and checks plan

previous chapter. With all of these pieces in place, the expert can now perform an impact analysis by deciding if the change plan creates a system that does not conform to the policies and modify the sequence of change operations until a suitable plan is developed (Figure 4.1). Once the expert has developed a plan, it can be saved, scheduled, and executed.

To be able to use the model and operations to provide a benefit for the expert, this process needs to be automated as much as possible and the language for expressing the plan needs to be defined. There needs to be software that can interpret the plan and check that during the execution the system does not violate its policies. Without such software, checking for policy violations in a plan becomes an arduous task, and planning in this style would have few or no benefits beyond what is currently done. Therefore, this chapter expands the model and operations into a full language and presents an interpreter for that language. Once the language is defined, an example of the planning workflow is given to provide a view of how such a way of planning would work.

### *4.1 Syntax and Semantics of Coeus*

Coeus is the language developed to manipulate the configuration model described in chapter 2 with the operations identified in the scenarios in chapter 3. It allows system administrators to specify application landscapes with policies; create subscriptions to landscapes; commission, change, and decommission instances inside and outside sub-

scriptions; specify behaviors for types; create and remove usage relationships; and query the model for various pieces of information.

A Coeus program consists of a list of declarations, statements, and expressions. Figure 4.1 provides a mostly complete abstract grammar of the language. In order to keep the language grammar presented here simple, distinctions between various types of expressions have been omitted[2].

Coeus is a dynamically typed language that does not require declaration of variables, supports subroutines (called "actions"), and automatically coerces data into an appropriate type for an operation when possible. The semantics of the type coercions are inspired by the Perl programming language, which has many of the same properties (see appendix A for a short explanation of Perl). For this reason the binary operators contain the standard mathematical operators (`+`, `-`, `/`, `*`, `>`, `<`, `>=`, `<=`, and `==`) as well as equivalent operators to operate on strings (concatenation `~`; lexicographic ordering `ge`, `le`, `gte`, `lte`; and equality `eq`, `ne`). Additionally, when an array is used with a binary or unary operation a coercion is performed which converts the array into the number that represents its length. Identifiers are sequences of alphanumeric and underscore symbols and must begin with a letter. Identifiers used for landscapes and types must begin with an uppercase letter in order to allow for disambiguation in cases where both a type name and a variable are allowed in the syntax.

Another feature of the language that is borrowed from the Perl language is the definition of truth. Any expression that evaluates to the number 0, the empty string, or a string that represents the number 0 are all false, and any other value is true. This definition of truth is useful for policies and behaviors where a truth value is sought, which, combined with the fact that arrays evaluate to their length in unary operators, means that in the context of policies and behaviors the expressions `?[` *e* `]` and `?[` *e* `]!= 0` are equivalent.

A final feature that is borrowed from Perl is that of "topics". The Perl language has a special variable `$_`, called the topic, that is used in several contexts (such as loops) to refer to the data that is to be operated on[3]. In Coeus the variable `_` is used in the same

[2]The concrete grammar makes a distinction between policy, behavior, boolean, and standard expressions.

[3]See [34] for a somewhat more philosophical and linguistic discussion of topics. Readers from a C++/Java background may be more comfortable comparing the topic to the `this` keyword, but

| | | |
|---|---|---|
| $bop$ | $\in$ | *BinaryOp* |
| $i$ | $\in$ | *Identifier* |
| $uop$ | $\in$ | *UnaryOp* |
| $t$ | $\in$ | *String* |
| $n$ | $\in$ | *Number* |
| $s$ | $\in$ | *SystemStatement* |
| $c$ | $\in$ | *ConfigurationStatement* |
| $d$ | $\in$ | *Declaration* |
| $e$ | $\in$ | *Expression* |
| $a$ | $\in$ | *Attribute* |
| $s$ | ::= | $d \mid c \mid$ “`include`” $t \mid$ “`subscribe`” “`(`” $i$ “`,`” $e$ “`)`” |
| | $\mid$ | “`unsubscribe`” “`(`” $e$ “`)`” $\mid$ “`in`” $e$ “`{`” $c^*$ “`}`” $\mid s$ <end of line> $s$ |
| $d$ | ::= | “`landscape`” $i$ “`{`” (“`*`”? $i$ $a$? “`:`” $e)^+$ “`}`” |
| | $\mid$ | “`type`” $i$ “`{`” $(i \mid e$ “`->`” $i)^*$ “`}`” $\mid$ “`action`” $i$ “`(`” $(i$ (, $i)^*)$? “`)`” “`{`” $s^*$ “`}`” |
| $a$ | ::= | “`(`” ( “`resilient`” “`=`” $n \mid$ “`comment`” “`=`” $s$ ) |
| | | ( “`,`” (“`resilient`” “`=`” $n \mid$ “`comment`” “`=`” $s$ ) )$^*$ “`)`” |
| $c$ | ::= | $e \mid$ “`policy[`” $i$ “`]`” $\mid$ “`start[`” $e$ “`]`” $\mid$ “`stop[`” $e$ “`]`” $\mid$ “`atomic`” “`{`” $c^*$ “`}`” |
| $e$ | ::= | $e$ *bop* $e \mid$ *uop* $e \mid s \mid n \mid i \mid i$ “`.`” $i \mid$ “`.`” $i \mid i$ “`(`” $(e$ (“`,`” $e)^*)$? “`)`” |
| | $\mid$ | “`![`” ( $i$ (“`:`” $i)$? $\mid e$ “`->`” $i\ e \mid e$ “`<-`” $i$ (“`:`” $i)$? ) “`]`” |
| | $\mid$ | “`X[`” $e$ “`]`” |
| | $\mid$ | “`?[`” ( $(i \mid$ “`?`”) (“`:`” $i)$? ( “`\|`” $e$ )? |
| | | $\mid (e \mid$ “`?`”) “`->`” $(i \mid$ “`?`”) $(e \mid$ “`?`”) ( “`\|`” $e$ )? |
| | | $\mid e$ “`<-`” $(i \mid$ “`?`”) (“`:`” $i)$? ( “`\|`” $e$ )? |
| | | $\mid (i \mid$ “`?`”) (“`:`” $i)$? “`<-`” $e$ ( “`\|`” $e$ )? ) “`]`” |
| | $\mid$ | $e$ “`{`” $c^*$ “`}`” |
| | $\mid$ | “`can?(`” $i$ “`)`” $\mid$ “`running?(`” $e$ “`)`” |

Table 4.1: Abstract Syntax of Coeus

fashion. For each expression of the language where the topic is relevant, what the topic refers to will be described during the explanation of the expression itself.

The evaluation environment for Coeus consists of a variable binding, the entire system, and the subscription that is being manipulated. Since a specific subscription is not always being manipulated (instead, the system's configuration as a whole can be), there may not always be a subscription in the environment. In addition to the environment, there is a derived property of the environment called the current configuration. The current configuration is the system's configuration if there is no subscription being manipulated or the subscription's configuration if there is a subscription being manipulated. The structure and behavior of the evaluation environment is important for understanding how commissioning and decommissioning are handled for subscriptions and the system.

### *4.1.1 Type Declarations and Calculating Capabilities*

The "`type`" $i$ "`{`" $(i \mid e$ "`->`" $i)^*$ "`}`" declaration is used to create a new type named $i$, which must begin with a capital letter, along with the associated behavior. The behavior, $b \in \mathit{Inst} \times \mathit{Conf} \rightarrow \mathcal{P}(\mathit{Label})$, is composed of the set of capability expressions (the expressions between the curly braces), which take two different forms: $i$ and $e$ "`->`" $i$. The first form is an *unconditional* capability expression, meaning that the value of the behavior function will always contain this capability.

$$\forall n \in \mathit{Inst}, c \in \mathit{Conf} : i \in b(n, c)$$

The second form is a *conditional* capability expression. The capability $i$ will only be in the value of the behavior function if the expression $e$ evaluates to a true value. In the evaluation of the expression the topic is bound to the instance for which the capabilities are being computed, which provides access to the parameters of the instance. The expression is evaluated in the scope of the system's configuration, so that the expression has access to all instances and usages.

Expressions used for conditional capabilities also have two restrictions on them. The first restriction prevents the use of any expressions that would cause a configuration

---

extended beyond the context of objects.

to change, such as assignment, commissioning, and decommissioning. The second restriction is that there are no variables allowed other than the topic variable. Since a capability expression is evaluated with only the topic variable bound and assignment is not allowed, all other variables would be undefined and therefore useless[4].

Calculating the capabilities of the instances in a system is a straightforward task for the most part. The capabilities of each individual instance are calculated repeatedly until the capabilities of the instances stop changing. Algorithm 1 shows how to calculate the capabilities of a single instance. Because there is the possibility of not only adding but

**Algorithm 1** Calculating capabilities for an instance

**Require:** $i \in Inst$

1: Clear the set of capabilities *caps* for $i$
2: Add all unconditional capabilities to *caps*
3: $seen \leftarrow \{caps\}$
4: **repeat**
5: $last \leftarrow c$
6: **for all** $(e, cap)$ in the conditional capabilities for $i$ **do**
7: **if** $e$ evaluates to **true then**
8: add *cap* to *caps*
9: **else**
10: remove *cap* from *caps*
11: **end if**
12: **end for**
13: **if** $caps \in seen$ **then**
14: **return false**
15: **end if**
16: $seen \leftarrow caps \cup seen$
17: **until** $last = c$

also removing capabilities, the calculation for instance (as well as system) capabilities is not guaranteed to find a set of capabilities. A simple example that illustrates this

[4]This restriction could be loosened to prohibiting assignment to parameters of instances while still allowing assignment to simple variables, since an assignment to a variable does not change the configuration.

Listing 4.1: Instance for which the capabilities cannot be calculated

```
type A {
       not can?(x) -> y
       can?(y) -> x
}
```

problem can be found in listing 4.1. The example uses the "`can?(`" *i* "`)`" expression which returns true if *i* is in the capabilities of the current topic. On the first iteration through, *caps* will contain both `x` and `y`. The next iteration will remove `y` because *caps* contains `x` and then will remove `x` because *caps* no longer contains `y`. This pattern then continues and no set of acceptable capabilities is found. Even if a set of capabilities is found, there is no guarantee that it is unique. If the capability expressions of a type are understood in terms of a logic program, then it can be seen why the given algorithm cannot always find a unique set of capabilities for any given instance. The behavior given in listing 4.1 represents a *non-stratified*[5] logic program. The method of calculating the capabilities given in algorithm 1 is simply a fixed point calculation of the capability expressions. Unfortunately, for non-stratified logic programs with negation there is not guaranteed to be a unique minimal stable model (meaning that the instance could have a different set of capabilities than what is found, and the other set is just as acceptable), but if the capability expressions are limited to stratified logic programs, then the iterated fixed point algorithm will find a unique stable model [21].

### *4.1.2 Commissioning, Decommissioning, and Lifecycle Control*

Commissioning has three different forms, depending on what is being commissioned: a single instance, a usage relationship, or an instance hosted on an existing instance. The semantics of each form of commissioning depends on whether there is a subscription in the environment or not. When there is a subscription in the environment then commissioning affects both the subscription's configuration as well as the system's configuration.

[5] A logic program is defined as a program $LP = T_0 + T_1 + \ldots + T_n$, where $T_k$ is formed of clauses $S \rightarrow p$ where $S$ is a, possibly empty, set of literals and $p$ (the conclusion) is a propositional letter. A stratified logic program is a logic program such that if $p$ belongs to the conclusion of a clause in $T_k$ then $p$ does not belong to $T_0 \ldots T_{k-1}$ nor does $\neg p$ belong to $T_0 \ldots T_k$ [20].

Listing 4.2: Commissioning and sharing examples

```
![A] # Commission an instance of A into the system
in 'a' { ![A:uniqueId] } # Commission another instance into
                         # the system and subscription a
in 'b' { ![A:uniqueId] } # Commissions the same instance
                         # into subscription b
![ ![A] -> foo ![A] ] # commission two instances and create a usage
                      # relationship between them
![ ![A] <- A ] # commission an instance and install another
               # instance on it
```

Decommissioning has these differing semantics as well, but the syntax is much simpler.

The simplest form of commissioning is for an instance of a given type and is done with the "`![`" *i*("`:`" *i*)? "`]`" expression. The instance commissioning expression creates a new instance of the type given by the first identifier and can optionally specify a unique identifier for the new instance. The new instance is placed in the system's and the current subscription's configuration. If the unique identifier is specified, then no new instance will be created if there is already an instance with that identifier, but the instance will still be placed into the configurations, which allows for sharing instances between subscriptions by creating an instance with a known identifier and then commissioning it into multiple subscriptions. Listing 4.2 shows an example of commissioning and sharing an instance between two subscriptions "a" and "b".

The only other way of commissioning instances is by commissioning an instance with a hosting relationship with the "`![`" *e* "`<-`" *i*("`:`" *i*)? "`]`" expression. The expression is made of two parts: the hosting target expression and the instance type and unique identifier. The hosting target expression *e* is evaluated and provides one or more instances that will be the hosting instances. For each of the hosting instances a new instance of the provided type is commissioned and a hosting relationship is created between the commissioned instance and the hosting instance. Because of the No Multi-hosting property of configurations, there can only be one hosting instance provided by the hosting target expression if a unique identifier is provided for the new instance.

The final form of commissioning is for usage relationships (except for hosting relationships) with the "`![`" *e* "`->`" *i e* "`]`" expression. The first expression provides the users of the capabilities specified by the identifier, which comes from the providers given by the second expression. A usage relationship is created for each (user, provider) pair. In order to maintain the constraints on configurations, creating a usage relationship can end up causing instances to be placed in subscription configurations in addition to the usage relationships that are created.

Once an instance or usage relationship has been commissioned, its lifecycle can be controlled. The instance can be taken to the running state with the "`start[`" *e* "`]`" statement and to the stopped state with the "`stop[`" *e* "`]`" statement. To determine what lifecycle state an instance or usage relationship is in there is the "`running?(`" *e* "`)`" expression, which will return true when the instance or usage relationship is in the running state.

The syntax for decommissioning instances and usage relationships is much simpler than that of commissioning. Decommissioning is handled by the "`X[`" *e* "`]`" expression. The expression *e* is evaluated to provide the instances or usage relationships that are to be decommissioned. Just as with the commissioning of instances and usage relationships, decommissioning has different semantics depending on whether there is a subscription in the environment or not and whether it is an instance or a usage relationship that is being decommissioned. When there is no subscription in the environment, then the decommission removes the instances from the system configuration as well as from all subscription configurations. In the case where there is a subscription in the environment, then an instance will only be removed from the subscription and not from the system as a whole. Usage relationships are always completely removed from the system regardless of a subscription being in the environment.

### *4.1.3 Querying*

Querying the current configuration for instances and relationships is probably the most important activity in change planning. Without queries it is very hard to determine what the configuration contains and which parts will need to be changed. Coeus, therefore, contains queries for instances, usage relationships, and hosting relationships that have a syntax that follows that of commissioning and can be composed to perform just about

Listing 4.3: Instance query examples

```
?[A] # find all instances of type A
?[? | .attr == 2] # find all instances
                  # where the parameter "attr" is 2
?[?[A] <- ? | running?(_)] # find all running instances
                           # hosted on an instance of type A
```

any query of the configuration that may be needed.

The simplest query is for instances with the "`?[`" $(i \mid$ "`?`"$)$ $($"`:`" $i)?$ "`]`" expression. This will return all instances in the current configuration with the given type and optionally with the given identifier. If the type is given as "`?`" then all types will match the query; therefore, `?[?]` will return all instances in the current configuration.

The instance query is expanded to provide a query for hosted instances with the "`?[`" $e$ "`<-`" $(i \mid$ "`?`"$)$ $($"`:`" $i)?)$ "`]`" expression. This query will return all instances that match the type and unique identifier, if given, that are hosted on the instances from the $e$ expression. In order to find the instances that are *hosting* other instances the second form of the query can be used: "`?[`" $(i \mid$ "`?`"$)$ $($"`:`" $i)?)$ "`<-`" $e$ "`]`".

Querying for usage relationships is done with the "`?[`" $(e \mid$ "`?`"$)$ "`->`" $(i \mid$ "`?`"$)$ $(e \mid$ "`?`"$)$ "`]`" expression. The usage relationships returned are those used by any instances returned by the first expression for the given capability from any of the instances returned by the second expression. Any one of the parts of the query can also be "`?`", which will match anything; therefore, `?[? ->? ?]` will return all usages in the current configuration.

All queries can be augmented with a filter to further limit the instances returned. The filter is specified after the main part of the query with a pipe (`|`) and then an expression. For each of the objects that match the main part of the query the expression is evaluated with the topic variable set to the object. If the expression evaluates to true, then the object will be returned from the query. Listing 4.3 contains some examples of queries.

### *4.1.4 Landscape Declarations and Checking Policies*

The "`landscape`" $i$ "`{`" ("`*`"? $i$ $a$? "`:`" $e)^+$ "`}`" declaration creates a landscape named $i$ and its policies. A landscape must always have at least one policy and exactly one

default policy (marked with an asterisk "`*`"). Each policy is composed of a name, optional attributes, and an expression.

The expression part of a policy defines the policy itself. Unlike the conditional capability expressions for types, the policy expressions in landscapes are evaluated in the scope of a subscription's configuration. Therefore, the expression only has access to the instances and usages that are part of the subscription.

Much like the expressions used in declaring types, the expressions used in policies are not allowed to change the configuration and do not have access to variables, but since there is not any meaningful topic[6], policy expressions do not have the topic variable. Policies can, however, be composed by referencing other policies within a policy expression. A policy reference is an identifier inside a policy expression, where the identifier is the name of some other policy in the same landscape.

In addition to the policy expression, a policy can also have two attributes specified. The first attribute is a `comment` which simply provides some documentation and a more readable message when a policy violation is found. The second attribute, `resilient`, specifies the minimum *resiliency* of the policy. The `resilient` attribute expands the determination of conformance to be not only based on the configuration directly but also all derived configurations from a failure of up to a minimum number of instances and usages. If the `resilient` attribute is not specified then the minimum resiliency defaults to 0.

The `resilient` attribute is described as the *minimum* number of failures because, in order to handle composing policies, it is only used as the baseline for deciding the number of failures. The actual resiliency of a policy is calculated by the function res : $Expression \times Identifier \rightarrow Number$. In order to define res we need two more functions: minres : $Identifier \rightarrow Number$, which provides the minimum resiliency of a policy, and expr : $Identifier \rightarrow Expression$, which provides the policy expression of a given policy.

[6]The subscription could be considered the topic, but subscriptions have no parameters or other properties that can be accessed from Coeus.

Listing 4.4: Example of a landscape

```
landscape Foo {
        *start:
                other && ?[A]
        other(resilient = 2):
                ?[Bar | .baz == 2] >= 2
}
```

Using these two functions, res can be defined as:

$$\text{res}(e,p) = \begin{cases} \min(\text{res}(e_1,p), \text{res}(e_2,p), \text{minres}(p)) & \text{if } e = e_1 \texttt{ || } e_2 \\ \max(\text{res}(e_1,p), \text{res}(e_2,p), \text{minres}(p)) & \text{if } e = e_1 \ \textit{bop} \ e_2 \\ \max(\text{res}(e_1,p), \text{minres}(p)) & \text{if } e = \textit{uop} \ e_1 \\ \max(\text{res}(\text{expr}(e),e), \text{minres}(p)) & \text{if } e \in \textit{Identifier} \\ \text{minres}(p) & \text{otherwise} \end{cases}$$

Since landscapes and policy expressions are somewhat complicated it might help to look at an example and walk through the meaning of each part. Listing 4.4 contains a simple landscape declaration for a landscape named `Foo`. When a subscription is made for the landscape, the subscription will be using the `start` policy. The `start` policy requires that all configurations for the subscription have at least one instance of type `A` and conform to the `other` policy. The `other` policy requires that there are at least two instances of type `Bar` that have an attribute `baz` with a value of 2. The `start` policy does not declare any resiliency and so has a minimum resiliency of 0 (the default), whereas the `other` policy does declare a minimum resiliency of 2. Before checking a configuration for conformance to the `start` policy the actual resiliency needs to be calculated. Using

the definition given above for the actual resiliency one gets:

$$\begin{aligned}
\text{res}(\texttt{other \&\& ?[A]}, \texttt{start}) &= \max(\text{res}(\texttt{other}, \texttt{start}), \text{res}(\texttt{?[A]}, \texttt{start}), 0) \\
&= \max(\text{res}(\texttt{other}, \texttt{start}), \text{minres}(\texttt{start}), 0) \\
&= \max(\text{res}(\texttt{other}, \texttt{start}), 0, 0) \\
&= \max(\max(\text{res}(\text{expr}(\texttt{other}), \texttt{other}), 0), 0, 0) \\
&= \max(\max(\text{res}(\texttt{?[Bar | .baz == 2]>= 2}, \texttt{other}), 0), 0, 0) \\
&\vdots \\
&= \max(\max(\text{minres}(\texttt{other}), 0), 0), 0, 0) \\
&= \max(\max(2, 0), 0, 0) \\
&= 2
\end{aligned}$$

Therefore, when checking a configuration for conformance to the `start` policy all related configurations with up to two failures need to be checked as well.

The idea of configurations with failures has been mentioned already, but not yet fully explained. As a change plan is being checked, the operations that are in the plan "directly" cause changes to be made to the configuration of the system. In each of these "direct" configurations there are no failures. Given one of these direct configurations, when one instance or usage that is changed from a *FailState* of working to failed, creating a new configuration, this new configuration is said to have one failure. If that configuration then has one more instance or usage fail, then that new configuration is said to have two failures, and so on[7]. The question whether a direct configuration conforms to a policy of resiliency $x$ becomes a search through this set of the direct configuration plus failure configurations with up to $x$ failures[8].

In order to check that an entire system conforms, as defined in chapter 2, one needs to iterate over all combinations of instances and usages, derive a configuration from the direct configuration with the selected instances and usages marked as failed, and check

[7]Because a failure can cascade, the number of failures injected into the configuration and the number of failures that will actually be in the configuration are not the same. For instance, a server that is hosting software may fail which causes the software hosted on it to fail as well. This case would be considered to have only one failure even though there are more failures that have occurred.

[8]The failure modelling used here is a very simplified form of the failure analysis and modelling presented by Ortmeier in [33], which was the inspiration for performing the failure analysis. Expanding this failure modelling and analysis to fully encompass what Ortmeier presents may be a very useful extension.

that it conforms to all policies. The policy check defined in algorithm 2 implements this by checking that each individual subscription's direct configuration conforms to its policy (lines 1–5). The next step is to look through all of the possible failure combinations. Line 6 determines the size of the largest failure combination that needs to be examined. Line 7 then constructs a set of the various elements of the configuration that are candidates for failure. Notice that hosting usage relationships are excluded, since failing a hosting relationship is not a useful thing to check[9]. The next step is to check every combination of failures of instances and usages up to the maximum size (lines 8–20). This is done by cloning the direct configuration and failing each combination of instances and usages and then checking that the subscription configurations still conform to the policies.

There is one final point that needs to be addressed about policy expressions. When constructing policies it is quite common to want to formulate "for all" and "there exists". Although these two common mathematical expressions cannot be directly expressed in Coeus, there is a reformulation of them that can be. As documented in [38, section 4] the universal and existential quantifiers

$$\forall x \in X : p(x)$$

$$\exists x \in X : p(x)$$

can be reformulated as

$$\{x \in X | p(x)\} = X$$

$$\{x \in X | p(x)\} \neq \emptyset$$

respectively. This reformulation means that "for all" can be expressed in Coeus as `?[X | p(_)]== ?[X]`, and that "there exists" can be expressed as `?[X | p(_)]!= 0`. Using these formulations for universal and existential quantification most reasonable policies on the structure of configurations should be able to be expressed.

### *4.1.5 Subscriptions*

Subscriptions are not objects that are bound to variables or directly manipulable in the language. Instead a subscription is tracked by naming it with a string and using that

[9] In fact it is not even clear what the meaning of failing a hosting relationship is. Does it mean that an instance is suddenly not being hosted and is running on its own? Does it mean that the hosted instance has died? The reasonable meanings are all covered by instances and other usage relationships failing, and so the hosting relationships are not candidates for failure.

**Algorithm 2** Policy Checking Algorithm

**Require:** $s \in Sys$

**Require:** $r$ a mapping from subscription to the resiliency of the current policy

**for all** $\sigma \in Sys_{Sub}(s)$ **do**
  **if** $state(\sigma)(Sub_{Conf}(\sigma)) = \mathit{ff}$ **then**
    **return false**
  **end if**
**end for**
$m \leftarrow \max(\{\sigma \in Sys_{Sub}(s) \mid r(\sigma)\})$
$o \leftarrow Conf_{Inst}(Sys_{Conf}(s)) \cup \{u \in Conf_{Usage}(Sys_{Conf}(s)) \mid Usage_{Label}(u) \neq \text{hosting}\}$
**for** $n$ from 1 to $m$ **do**
  **for all** $f \in \binom{o}{n}$ **do**
    $s' \leftarrow$ clone $s$
    **for all** $e \in f$ **do**
      fail $e$ in $s'$
    **end for**
    **for all** $\sigma \in Sys_{Sub}(s')$ **do**
      **if** $n \leq r(\sigma)$ and $state(\sigma)(Sub_{Conf}(\sigma)) = \mathit{ff}$ **then**
        **return false**
      **end if**
    **end for**
  **end for**
**end for**
**return true**

name to refer to it later. This system works since there are only four statements that work with subscriptions, and since subscriptions themselves do not have any parameters or other properties that need to be dealt with.

Creating subscriptions to landscapes is done with the "`subscribe`" "`(`" *i* "`,`" *e* "`)`" statement. The identifier provides the name of the landscape for which to create the subscription, and the expression evaluates to a string that provides the name of the subscriptions. Removing a subscription is done with the "`unsubscribe`" "`(`" *e* "`)`" statement by providing an expression that evaluates to the name of the subscription to remove.

Once a subscription is in place, working with the configuration of the subscription is done with the "`in`" *e* "`{`" $c^*$ "`}`" statement. The `in` expression changes the current configuration to the configuration of the subscription given by *e* and executes the block of *ConfigurationStatement*s. After the block is done executing, the current configuration is returned to the system configuration.

Inside the block of an `in` expression the subscription's policy can be changed with the "`policy[`" *i* "`]`" statement. The subscription's policy is changed to the policy named by the given identifier, and the policy immediately takes effect.

### *4.1.6 Action Declarations, Calling, Topic Blocks, Atomic Blocks, and Including*

In order to facilitate plan reuse, Coeus also includes actions. Actions are nothing more than the procedures or subroutines that are in other languages. An action is declared by the "`action`" *i* "`(`" $(i\ (,\ i)^*)?$ "`)`" "`{`" $s^*$ "`}`" declaration. The first identifier provides the name of the action, and the rest of the identifiers provide the formal parameters for the action. The body of an action is composed of a sequence of statements.

When the action is called, using the common syntax of *i* "`(`" $(e\ (\text{“,”}\ e)^*)?$ "`)`", the expressions are evaluated and the values are bound to the variables for the parameters in the scope of the body of the action. The statements of the body are then evaluated, and the value of the last statement is provided as the return value. Listing 4.5 provides an example of a simple action declaration and action call.

Another form of reuse can be accomplished by creating files that contain plans that can be reused. The "`include`" *t* statement causes the file named *t* to be read in and interpreted in the same scope as the `include` statement itself is in.

Listing 4.5: Example of an action

```
action add(x, y) {
        x + y
}
x := add(2, 3) # x will be 5
```

Listing 4.6: Two examples of topic blocks

```
# modify an instance right after it is commissioned
# but before being assigned to x
x := ![A] {
        .attr := 1
}

# Stop and decommission all instances of type A
?[A] {
        stop[_]
        X[_]
}
```

At times one needs to specify a change plan that is composed of several steps, but can be executed on the actual systems as a single step. To allow specifying such multi-step yet atomic changes, there is the "`atomic`" "`{`" $c^*$ "`}`" statement, which suspends policy checking for the duration of the block to simulate a single step.

Finally, there is also the $e$ "`{`" $c^*$ "`}`" expression, which is called a topic block. The topic block iterates over the values of the expression and evaluates the statements in the block for each value. During each iteration through the block the topic variable is set to the current value, hence the name topic block. This allows describing plans where some set of changes are performed to instances or usages that are only known by querying the configuration for the information. Listing 4.6 shows some examples of topic blocks.

## *4.2 The Coeus Interpreter*

An implementation of the language was created to test its usefulness. The complete language is supported as well as a plugin facility that allows for hooking into the interpretation of the language to support debugging or various other extensions. Two extensions were created during the course of the implementation as need arose. The implementation itself is in the Perl programming language. Since Perl may not be known to all readers, an explanation of how to read Perl has been provided in Appendix A. Perl was chosen because of the author's familiarity with it and because of the closeness of the Coeus semantics and Perl semantics, which simplified the implementation immensely.

### *4.2.1 Implementing the Configuration Model and Changes*

The translation from the mathematical model presented in chapter 2 to a collection of classes presents the basis of the implementation. Figure 4.2 shows a UML class diagram of the configuration model that is used in the interpreter. For brevity, not all methods are listed on the classes, only the methods that help in understanding where responsibilities lie. The System class acts as the main interface for interacting with the rest of the model. It handles keeping its configuration and the subscription configurations synchronized with one another and delegates to other classes as appropriate. The Behavior class is the behavior function of a type, and although it is represented here as a class following the Command pattern, the actual implementation of it is an anonymous subroutine closure. A Policy on the other hand also has aspects of a simple Command pattern, but since a policy also needs to be able to calculate its resiliency (using the method presented earlier), it is implemented as a class.

The parser for the Coeus language was written using the Parse::RecDescent parser generator by Damian Conway. Parse::RecDescent compiles a grammar with actions to a parser that can be used to create an abstract syntax tree of Coeus source code. The AST generated by the parser is then evaluated in a manner similar to the technique described by Abelson et al. in [4, pg. 364–373]. For each type of node in the AST there is an evaluation function defined to handle it. The evaluation functions are split between special forms and normal forms (listing 4.7 shows the main evaluation function). Normal forms are AST evaluations where all of the child ASTs are first evaluated and then the

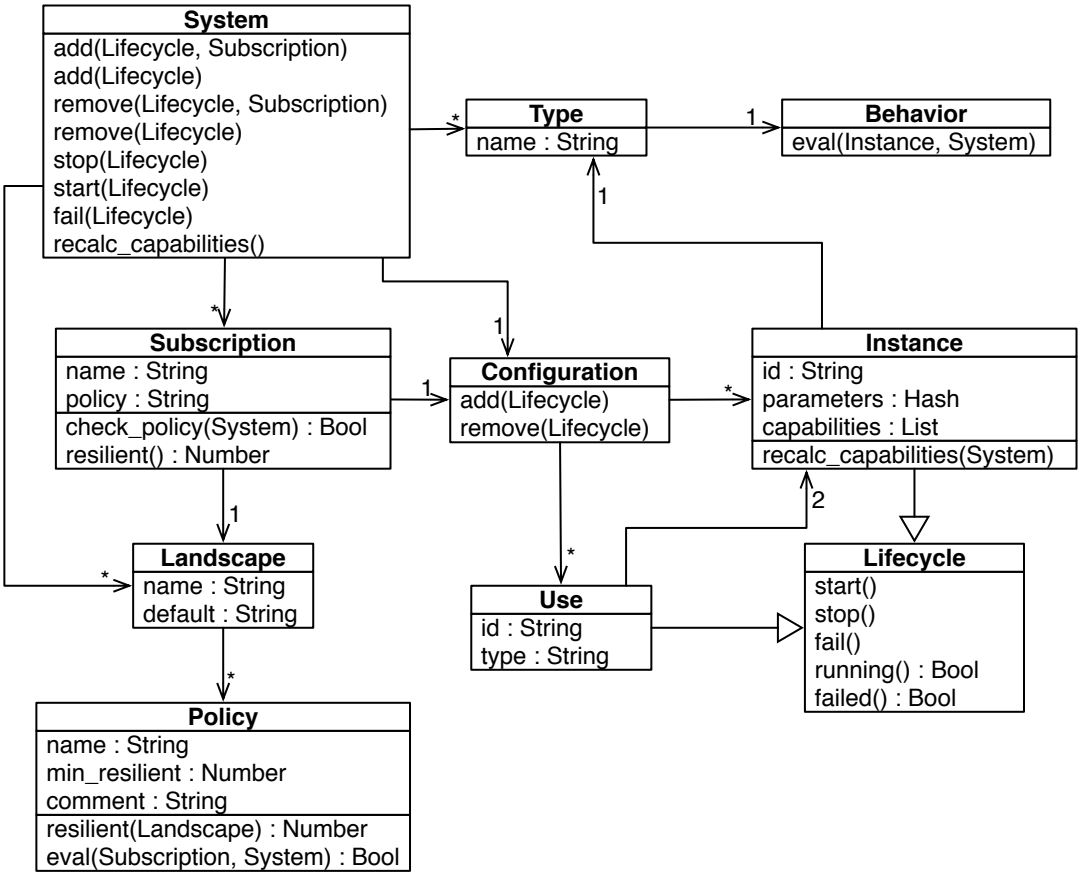



Figure 4.2: Class diagram of the configuration model

Listing 4.7: Evaluation of the Coeus AST

```
40 sub coeus_eval {
41     my ($expr, $env) = @_;
44     my $op = $expr->[0];
45     my @args = @$expr[1 .. $#$expr-1];
46     my $line = $expr->[-1];
51     my (@eval_args, $f);
52     if(exists $normal_forms{$op}) {
53         @eval_args = map { defined $_ ? coeus_eval($_, $env) : undef } @args;
54         $f = $normal_forms{$op};
55     }
56     elsif(exists $special_forms{$op}) {
57         @eval_args = @args;
58         $f = $special_forms{$op};
59     }
64     my $r = $f->($env, @eval_args, $line);
69     return $r;
70 }
```

node itself is evaluated (lines 53–54 and 64). A special form gives the control of the child ASTs to the evaluation function for the node (lines 57–58 and 64). As each node is evaluated the hooks into the interpretation are called. Hooks can be registered for before, after, or before and after the evaluation of a particular node (the hook calling code has been removed from the `coeus_eval` code listing for simplicity).

Policy checking is not a part of the interpreter itself. Instead, the policy checking code is a hook that executes after a node has been evaluated (listing 4.8). The semantics of the atomic blocks can be seen being implemented by the two hooks on lines 13–16 and the check done in the `_check_policies` subroutine on line 24. The `_find_counterexample` subroutine used on line 26 checks the conformance of the system using algorithm 2 and returns the failures that were found to cause a non-conform system as well as the subscription that did not conform. The interpretation is stopped by throwing an

Listing 4.8: Policy checking hooks

```
12 my $in_atomic = 0;
13 Coeus::Interpreter::Hook::register_entry(
14     sub { $in_atomic++ if $_[1] eq 'atomic_block' });
15 Coeus::Interpreter::Hook::register_exit(
16     sub { $in_atomic-- if $_[1] eq 'atomic_block' });
17
18 Coeus::Interpreter::Hook::register_exit(\&_check_policies);
19
20 sub _check_policies {
21     my ($env, $op) = @_;
22
23     return unless Coeus::Interpreter::AST::modifies_configuration($op);
24     return if $in_atomic;
25
26     my $counterexample = _find_counterexample($env->system, 0);
27     if ($counterexample) {
28         die "Configuration␣does␣not␣conform␣to␣policy\n"
29           . _stringify_counterexample($counterexample);
30     }
31 }
```

exception on line 28 when a counterexample is found. The text of the error contains the failures as well as the policies that were found to be violated (provided by the `policy_errors` method of the subscription).

### *4.2.2 Tracing and Visualizing*

In the course of writing the interpreter, examples, and tests it became clear that there needed to be a way to discover what was going on as a change plan was being checked. Two hooks were created to provide information about the state of the configuration and the execution of the plan. The first hook transforms the system and subscription

Listing 4.9: Example plan for visualization and tracing

```
type Server {
    .ram > 4*1024 -> x86_64
}
type Apache {
    ?[?[Server | .ram >= 512] <- _] && .perl -> mod_perl
}

![Server] {
    .ram := 256
}

![Server] {
    .ram := 5*1024
}

![?[Server] <- Apache] {
    .perl := 1
}
```

configurations into graphs that can be shown with the GraphViz program[10]. The second hook provides a trace of the execution of the plan by printing important operations to the console. Given the plan shown in listing 4.9 the tracing hook produces the output shown in listing 4.10. At the same time the visualization hook produces a dot file that can be graphed with GraphViz to provide a view of the configuration as shown in figure 4.3.

### *4.3 Planning Example*

In creating an example of planning using Coeus and the interpreter, it quickly became clear that a realistically sized system model poses several problems. Often real systems

[10]GraphViz draws directed or non-directed graphs that are described in the Dot language. It is available from `http://www.graphviz.org`.

Listing 4.10: Trace produced by listing 4.9

```
==> Entering commission
<== commission(Server) => Server:Instance_0
==> Entering bind
<== Server:Instance_0.ram := 256
==> Entering commission
<== commission(Server) => Server:Instance_1
==> Entering bind
<== Server:Instance_1.ram := 5120
==> Entering commission
<== commission(Apache) => Apache:Instance_2
==> Entering commission
<== commission(Apache) => Apache:Instance_3
==> Entering bind
<== Apache:Instance_2.perl := 1
==> Entering bind
<== Apache:Instance_3.perl := 1
```

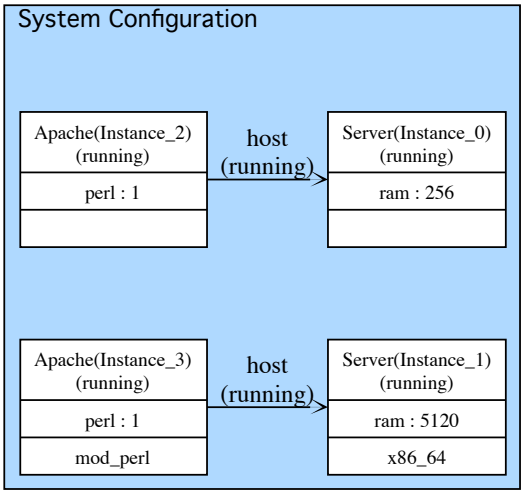



Figure 4.3: Visualization of the configuration produced by listing 4.9

are quite large (hundreds or thousands of instances), diverse (tens or hundreds of types), and interactions between the parts are quite complicated. This is of course the very motivation for creating Coeus, but for a simple example of planning with it a smaller system needs to be used. Therefore, the system that is presented here is specifically kept small to show the usage of Coeus in planning and not all of its capabilities.

The system is composed of two subscriptions, each to a different landscape, and several shared instances. A change plan needs to be created for one of the subscriptions. The example will show three attempts at constructing a change plan. Two of the attempts will show problems that can be uncovered during planning, and the final attempt shows a successful plan.

The example starts with the current configuration at the datacenter of a (very small) web hosting company (ExampleCom). The company has two customers. One customer only needs static web page serving but has a large amount of traffic and so has two web servers. The other customer has a small web application that receives very little traffic. In order to keep costs low, the hosting company has decided to share one of the servers between the two subscriptions. Figure 4.4 shows what the configuration currently looks like for the system as well as the two subscriptions. Listings 4.11 and 4.12 show the type and landscape definitions used.

The customer using the static web servers reports that the servers are no longer keeping up with traffic. They request that the servers be modified to increase their capacity. A system administrator at ExampleCom decides that the best way to achieve this is to add more memory to one of the servers and change both Apache2 processes to use a threaded multi-processing scheme instead of one based on forked processes.

### *4.3.1 First Attempt: An Immediate Problem with Policies*

The first change plan attempt is shown in listing 4.13. This plan changes one Apache2 instance to use the worker[11] MPM, and then stops the server it is on to add more memory. Once the server is back online it procedes to update and restart the second Apache2 instance.

The administrator then proceeds to check this plan using the Coeus interpreter.

[11] "This Multi-Processing Module (MPM) implements a hybrid multi-process multi-threaded server. By using threads to serve requests, it is able to serve a large number of requests with fewer system resources than a process-based server."[9]

Listing 4.11: The types used in ExampleCom's Coeus model

```
type Server { }
type Apache2 {
    ?[_ -> perl ?[Perl | can?(ithreads) && .version >= 5.8]] ->
        threaded_perl
    ?[_ -> perl ?[Perl | .version >= 5.8]] -> unthreaded_perl
    .mpm eq "worker" -> Threads
    .mpm eq "prefork" -> Fork
    .mod_perl && ((can?(Threads) && can?(threaded_perl)) || (can?(Fork)
        && can?(unthreaded_perl))) -> mod_perl
}
type VHost { }
type Perl {
    perl
    .has_ithreads && .has_multiplicity -> ithreads
}
```

Listing 4.12: The landscapes used in ExampleCom's Coeus model

```
landscape Static {
    *setup: 1
    production(resilient = 1): maintenance
    maintenance:
        ?[VHost | running?(_)] >= 1
}
landscape Dynamic {
    *setup: 1
    production:
        ?[Apache2 | can?(mod_perl) && running?(_)] && ?[VHost]
    maintenance: 1
}
```

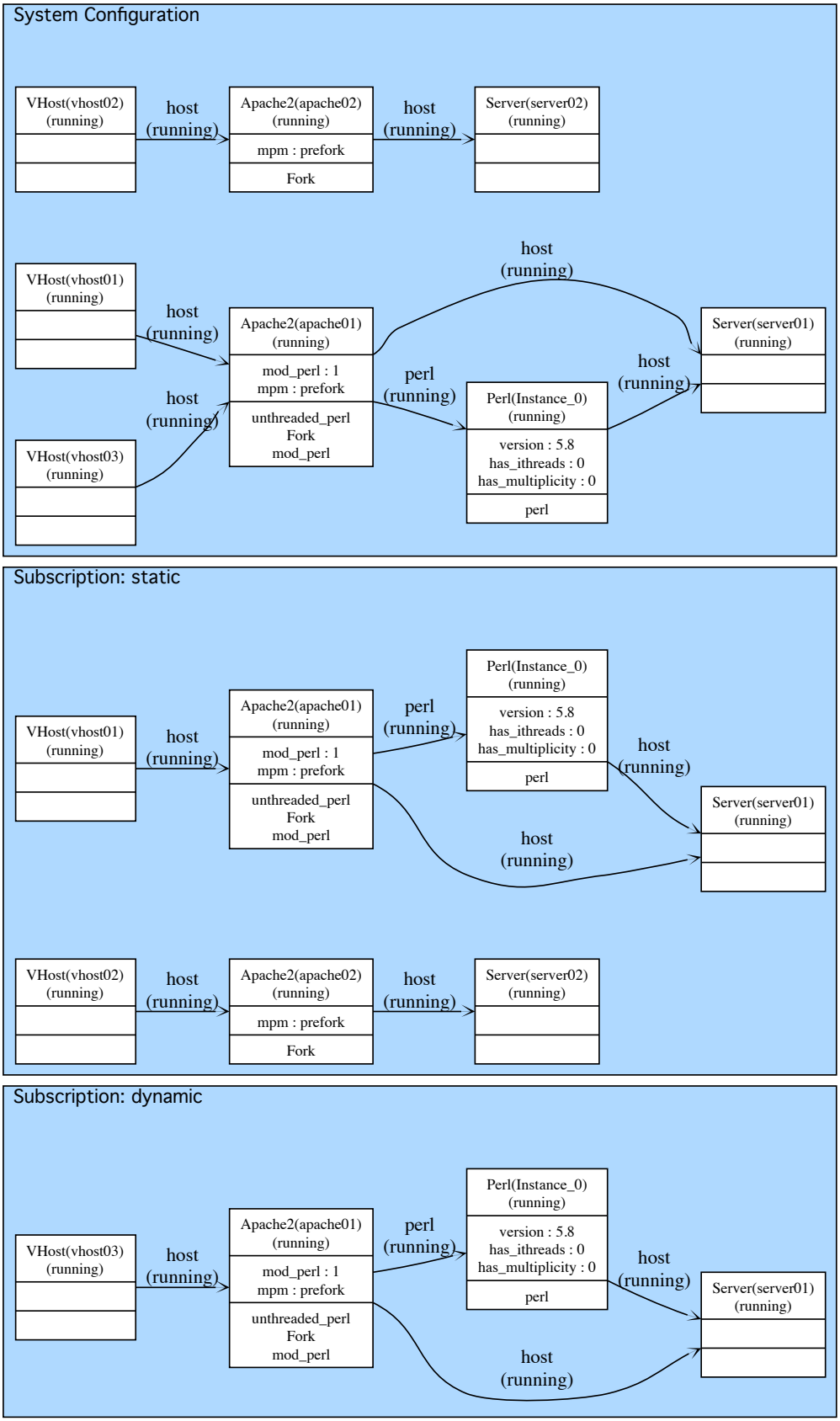



Figure 4.4: Starting configuration of the datacenter

Listing 4.13: First change plan attempt

```
in 'static' {
    ?[Apache2:apache02] {
        .mpm := "worker"
    }

    server := ?[Server:server02]
    stop[server]
    server { .memory := .memory + 1024 }
    start[server]

    ?[Apache2:apache01] {
        .mpm := "worker"
        stop[_]
        start[_]
    }
}
```

The interpreter, along with the trace hook, produces the output seen in listing 4.14. Coeus found that after stopping the `Server:server02` instance the configuration no longer conforms to the production policy (by way of the production policy using the maintenance policy, of which the production policy simply requires a higher resiliency) for the static subscription. By examining the configuration shown in figure 4.5 and the landscape definitions in listing 4.12 it is easily seen that if the `Apache2:apache01` instance were to fail the policy expression would no longer be true.

### *4.3.2 Second Attempt: Constraints from other Subscriptions*

The administrator has two choices: either the static subscription can be placed into its maintenance policy or more instances can be commissioned to keep the the subscription in conformance with its production policy as the changes are made. Since ExampleCom is such a small company, they have no extra resources that they can put to use, and so

Listing 4.14: Output from the first change plan attempt

```
 ==> Entering bind
 <== Apache2:apache02.mpm := worker
 ==> Entering bind
 ==> Entering stop_instance
 <== stop_instance((Server:server02))
ERROR: Configuration does not conform to policy
Counterexample:
        -> Instance(Apache2:apache01)
failed in the policy maintenance
failed in the policy production
for subscription static
```

the administrator must plan a maintenance window for performing the changes. The administrator can, however, make the window smaller by only placing it around the time that the server is shut down.

The second version of the change plan can be seen in listing 4.15. Just before the server is stopped the subscription is placed into its maintenance policy and returned to the production policy after the server is running again. This plan, however, also has problems as shown by the trace in listing 4.16. Interestingly, this time the problem does not come from stopping an instance, rather it comes from changing the MPM on the other Apache2 instance to use the worker MPM. The trace also shows that this time it is not a problem with the static subscription but with the dynamic subscription.

The problem can be seen by examining the configuration that was reached shown in figure 4.6, the types, and the landscapes. The dynamic landscape requires at least one Apache2 instance that has mod_perl and is running. The configuration shows that the Apache2 instance in the dynamic subscription no longer has the mod_perl capability. By examining the Apache2 behavior it becomes clear that there is not an appropriate Perl available to provide mod_perl when the worker MPM is being used[12].

[12]The behavior in the Apache2 type is based upon the information available from [10].

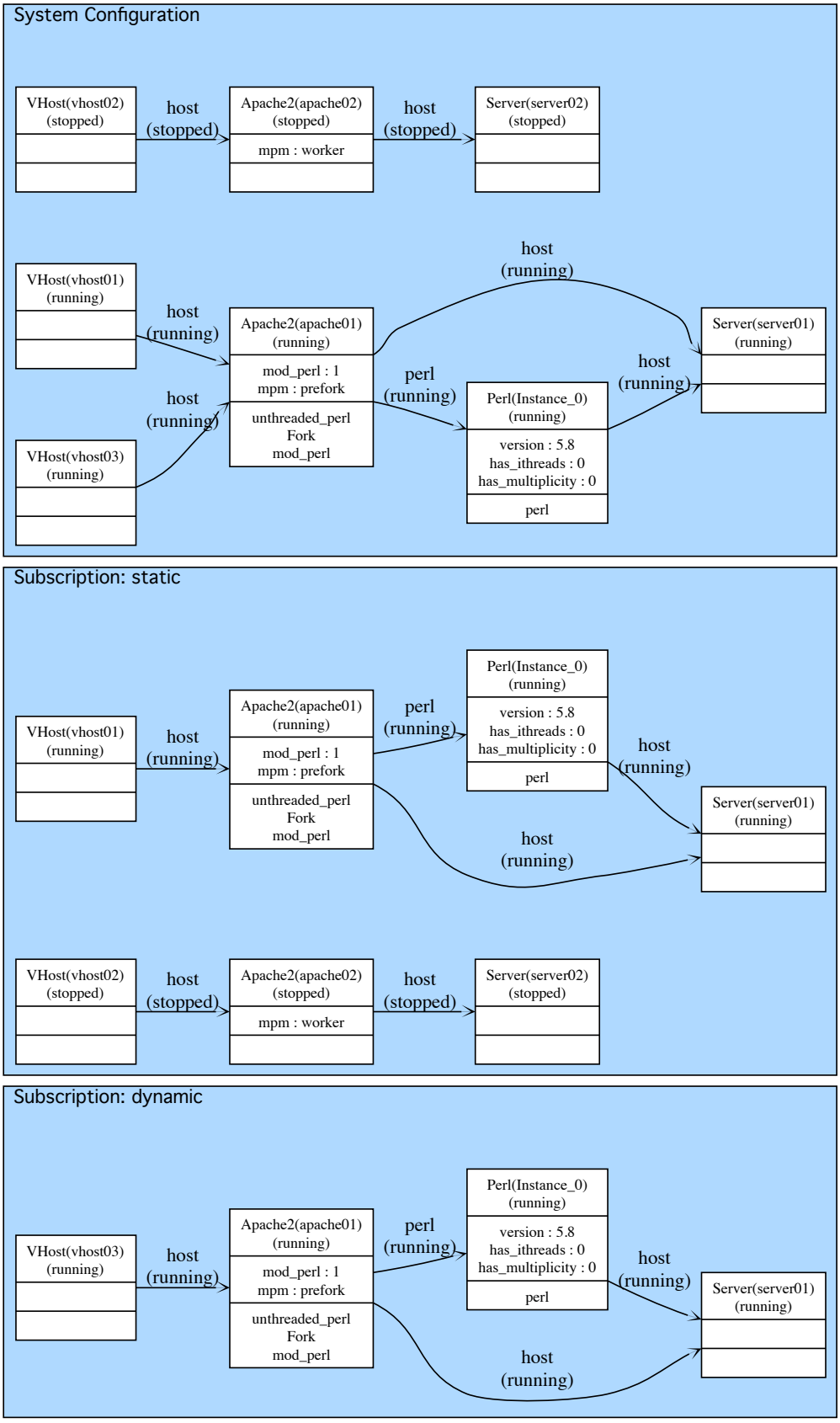



Figure 4.5: Configuration reached by the first change plan

Listing 4.15: Second change plan attempt

```
in 'static' {
    ?[Apache2:apache02] {
        .mpm := "worker"
    }

    server := ?[Server:server02]
    policy[maintenance]
    stop[server]
    server { .memory := .memory + 1024 }
    start[server]
    policy[production]

    ?[Apache2:apache01] {
        .mpm := "worker"
        stop[_]
        start[_]
    }
}
```

Listing 4.16: Output from the second change plan attempt

```
 ==> Entering bind
 <== Apache2:apache02.mpm := worker
 ==> Entering bind
 ==> Entering stop_instance
 <== stop_instance((Server:server02))
 ==> Entering bind
 <== Server:server02.memory := 1024
 ==> Entering start_instance
 <== start_instance((Server:server02))
 ==> Entering bind
 <== Apache2:apache01.mpm := worker
ERROR: Configuration does not conform to policy
Counterexample:
       No failures needed
failed in the policy production
for subscription dynamic
```

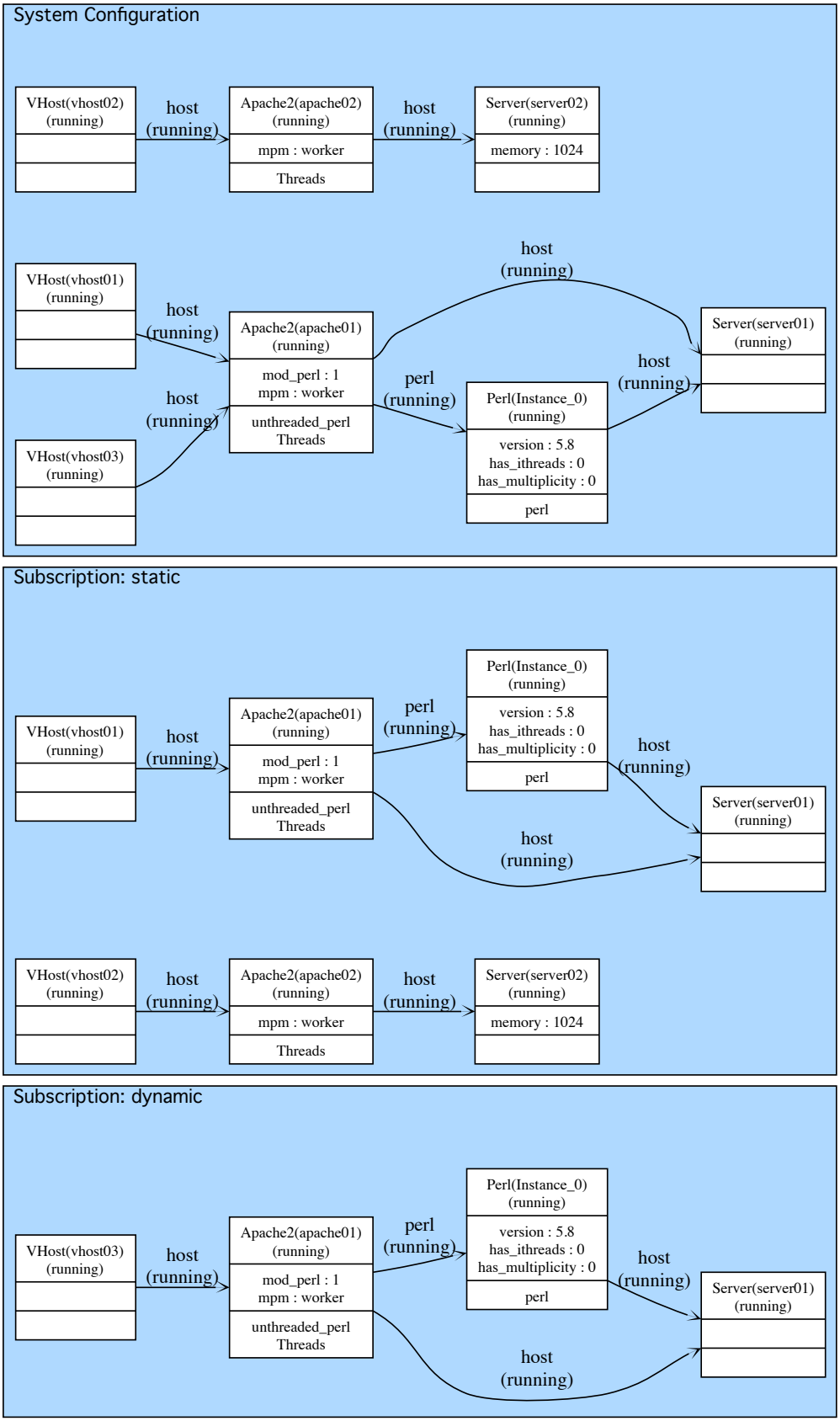



Figure 4.6: Configuration reached by the second change plan

### *4.3.3 Third Attempt: A Successful Change Plan*

What had originally been a change to the instances of a single subscription has now grown into a change plan that will need to affect two subscriptions. The administrator still needs to fulfill the customer's request of increasing capacity and, for cost reasons, continue to share the Apache2 instance with mod_perl between the two subscriptions. For Apache2 to work with mod_perl in a threaded mode there needs to be a Perl that supports threading.

The new plan to achieve this is shown in listing 4.17[13]. The new change plan performs the first part of the change to the static subscription (but does not return to the production policy), and then needs to switch to the dynamic subscription. In the dynamic subscription a new Perl that supports threads is installed on the server. Then the subscription needs to be changed to its maintenance policy and modify the Apache2 instance to use the new Perl. The old Perl can now be decommissioned and the static subscription returned to its production policy. Listing 4.18 shows that this plan conforms to the policies and figure 4.7 shows the final configuration that it will produce.

[13]It actually took me several more attempts to construct the final plan, but presenting them all would be a little long winded for a simple example of how to use the tool.

Listing 4.17: Successful change plan

```
include 'datacenter.coeus'
in 'static' {
    ?[Apache2:apache02] {
        .mpm := "worker"
    }
    server := ?[Server:server02]
    policy[maintenance]
    stop[server]
    server { .memory := .memory + 1024 }
    start[server]
}
in 'dynamic' {
    new_perl := ![?[Server] <- Perl] {
        .version := 5.8
        .has_ithreads := 1
        .has_multiplicity := 1
    }
    policy[maintenance]
    ?[Apache2:apache01] {
        .mpm := "worker"
        X[?[_ -> perl ?[Perl]]]
        ![_ -> perl new_perl]
        stop[_]
        start[_]
    }
    policy[production]
}
X[?[Perl | not .has_ithreads]]
in 'static' {
    policy[production]
}
```

Listing 4.18: Output from the successful change plan

```
==> Entering bind
<== Apache2:apache02.mpm := worker
==> Entering bind
==> Entering stop_instance
<== stop_instance((Server:server02))
==> Entering bind
<== Server:server02.memory := 1024
==> Entering start_instance
<== start_instance((Server:server02))
==> Entering bind
==> Entering commission
<== commission(Perl) => Perl:Instance_1
==> Entering bind
<== Perl:Instance_1.version := 5.8
==> Entering bind
<== Perl:Instance_1.has_ithreads := 1
==> Entering bind
<== Perl:Instance_1.has_multiplicity := 1
==> Entering bind
<== Apache2:apache01.mpm := worker
==> Entering decommission
<== decommission(([Apache2:apache01 -> perl Perl:Instance_0]))
==> Entering use
<== use(Apache2:apache01, Perl:Instance_1, perl) => [Apache2:apache01
   -> perl Perl:Instance_1]
==> Entering stop_instance
<== stop_instance(Apache2:apache01)
==> Entering start_instance
<== start_instance(Apache2:apache01)
==> Entering decommission
<== decommission((Perl:Instance_0))
```

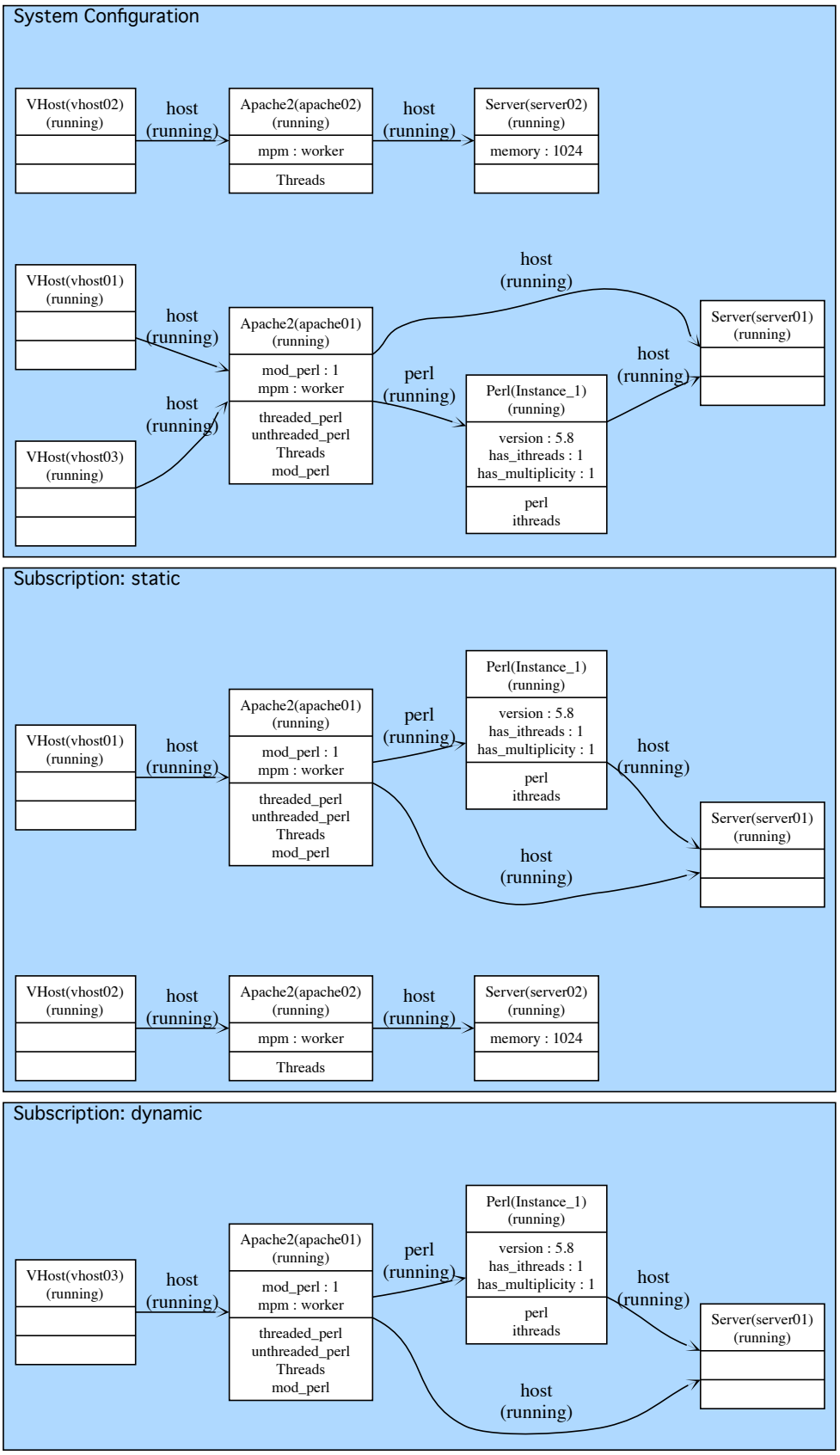



Figure 4.7: Target configuration reached

# Chapter 5

# CONCLUSION AND FUTURE DIRECTIONS

Currently, system administrators face the problem of ever increasing complexity in the configurations that they must manage. Interactions between components in these large, interdependent systems become hard to foresee and can easily lead to incorrect configuration change choices [31]. One solution to this problem lies in offloading the checking of interactions to a computer, freeing the human system administrator to deal with the larger problem of deciding what needs to be done. The language and model presented in this dissertation are a first step in providing system administrators with the tools that they need to implement this approach. The Coeus language allows for describing how to change the model, and the model provides a way for defining policies and instances.

Policies are defined in landscapes which can be subscribed to. Once a subscription to a landscape has been made, its current policy must hold for all of the instances that are used in it. Furthermore, instances can be shared among several subscriptions and therefore be subject to many different, and competing, policies. This modelling of policies provides a way for expressing the many different requirements that a single instance in a larger system must fulfill.

In order to be able to express the interactions in a configuration, instances are more than simple collections of key-value pairs to hold configuration parameters; they also have a defined behavior that interprets their parameters and environment into a set of capabilities. The capabilities can then be used in policies to determine if the system provides the services that are required of it or in the behavior of other instances to drive more capabilities.

The behaviors, policies, commissioning, decommissioning, and various other actions are all expressed in the Coeus language. The interpreter for the Coeus language reads a definition of types and landscapes and a sequence of change commands and determines if throughout the execution of the commands the policies continue to be fulfilled. In order to support construction of a change plan, the interpreter also provides a visualization

of the configuration, a trace of the commands executed, and an explanation of how a policy is not fulfilled.

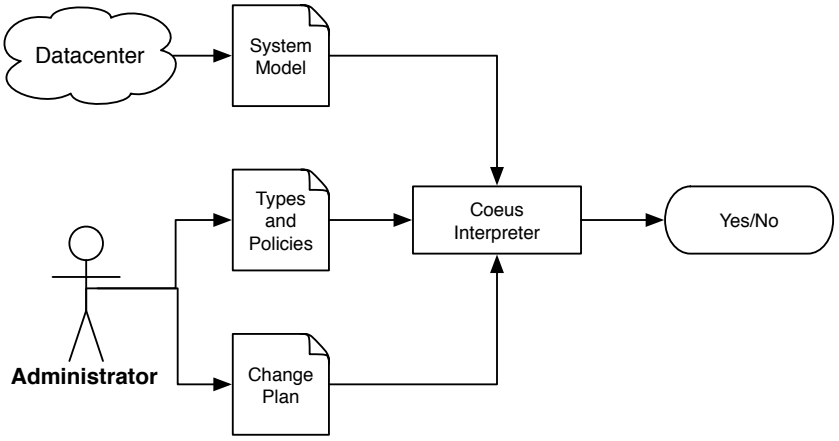



Figure 5.1: Using Coeus to check change plans

A system administrator can use the tool developed in this dissertation to check change plans. By adding this as a formal validation step to change plans in the ITIL process, the number of service failures due to misconfiguration should be reduced. The administrators no longer need to keep all of the interconnections of their application landscape in their head to search for failures and can instead rely on the computer to ensure that the proposed change does not cause a failure.

The Coeus language, however, needs to be used in real situations to discover where it lacks the expressiveness needed to describe large, complicated, and heterogeneous systems. Attempting to use the language in real scenarios would also provide information about what kinds of questions about configuration changes need to be answered that it does not currently address. Currently, Coeus is only able to provide a yes or no determination about a change, but it might be useful to be able to provide additional information and receive quantitative information about change plans.

Examples of quantitative information that could be useful when applied to change operations include time, cost, and failure likelihood. If such quantitative information were to be combined with enhanced policies (such as allowed non-compliance for policies to express such things as service level agreements of 90% uptime), then an administrator

would be able to use the language for selecting between several acceptable plans.

Another improvement would be to allow the use of temporal logics (such as LTL or CTL) in specifying policies. Using temporal logics would allow analysis much similar to that defined in [33]. They would also require a model checking algorithm to be used for determining policy compliance instead of the naive search that is currently used.

The configuration model used by Coeus is inadequate to faithfully model many systems. The current lifecycle model used does not have enough flexibility to represent instances that have different relationships to their hosting instances, thereby preventing much of the complexity of stopping and starting instances from being modelled. By using a model that is much closer to that used by the SmartFrog system (see [23, pg. 112–114] for an explanation of how lifecycles in SmartFrog are handled), much more flexibility would be available.

Another shortcoming in the Coeus configuration model is its inability to clearly tie a capability of an instance to the parameters of the instance. For instance, Coeus can express that an instance has the capability to provide HTTP but cannot easily express that the HTTP is only available on a particular port. By being able to parameterize the capabilities and usage relationships, much more information about effected systems would be available. This would also increase the expressiveness of the behaviors to be similar to that of logic programming languages and may require that they are interpreted in the fashion of logic programming languages as well.

The accuracy of the analysis performed by Coeus is, of course, entirely dependent upon the quality of its configuration model and the fidelity in the execution of the plan. If the creation of the initial configuration model were in some way automated, then the quality of analysis from it would be much more certain. The IBM Change and Configuration Management Database (see [28] for a description of the IBM CCMDB) provides much or all of the configuration information that would be required to populate the initial configuration model. It may be useful to provide a way for Coeus to either use the IBM CCMDB directly or define a translation from the IBM CCMDB to the model used by Coeus.

Once the change plans have been created in Coeus they need to be executed correctly. Executing the plans and ensuring that every step happens at the correct time and in the correct order is an error prone task. An interpreter of Coeus that can execute the

changes on a configuration engine (such as Bcfg2 or LCFG) could be used to automate the execution of the plan. If the plans could be executed directly, then many human errors can be removed from the equation and the correctness of the configuration change can be increased.

In addition to expanding the Coeus language and interpreter, there are also a great many interesting directions to investigate from the standpoint of formally modelling configurations. Areas such as modelling security aspects of a configuration (such as firewall rules) or allowing composition of subscriptions (thereby better modelling service oriented architectures) present interesting problems.

Investigating configurations even more formally through the use of graph grammars could also be useful. The Coeus configuration model is essentially a directed graph, and therefore the changes can be seen as graph rewritings. The authors of [35, pg. 192] explain:

> Among the many formalisms that can be chosen to represent distributed systems and their evolutions, we believe that graphs and graph grammars are among the most convenient, both in terms of expressivity and of technical background. In fact, graphs describe in a natural way net topologies and data sharing, and moreover they possess a wide literature and technical results which make the whole field of graph rewriting very formal and its notions precisely definable. ... the evolution of such system [sic] can be modelled by graph rewriting: at each rewriting step a subpart of the current graph is chosen to evolve, and its evolution consists of being replaced by another graph.

These are some of the directions in which Coeus could be taken, and the author hopes that these possibilities will be investigated. As the world becomes more and more dependent on internet services for communication, knowledge retrieval, entertainment, publishing, and so many other human activities, the importance of these highly complicated computer systems grows. The future will surely lie in the direction of fully autonomous systems, but until that future arrives the system administrators tasked with maintaining these important services need the tools with which to reason about, build, and maintain the foundation of our modern information society.

## BIBLIOGRAPHY

[1] *Proceedings of the 17th Conference on Systems Administration (LISA 2003), October 26-31, 2003, San Diego, California, USA* (2003), USENIX.

[2] *Proceedings of the 19th Conference on Systems Administration (LISA 2005), December 4-9, 2005, San Diego, California, USA* (2005), USENIX.

[3] *Proceedings of the 20th Conference on Systems Administration (LISA 2006), December 3-8, 2006, Washington, DC, USA* (2006), USENIX.

[4] Abelson, H., and Sussman, G. J. *Structure and Interpretation of Computer Programs, Second Edition*. MIT Press, 1996.

[5] Anderson, P. A declarative approach to the specification of large-scale system configurations. `http://homepages.inf.ed.ac.uk/dcspaul/publications/conflang.pdf`.

[6] Anderson, P. Why configuration management is crucial. *;login: 31*, 1 (February 2006), 5–8.

[7] Anderson, P., Goldsack, P., and Paterson, J. Smartfrog meets lcfg: Autonomous reconfiguration with central policy control. In *LISA* [1], pp. 213–222.

[8] Anderson, P., and Smith, E. Configuration tools: Working together. In *LISA* [2], pp. 31–37.

[9] Apache Software Foundation. Apache MPM worker. Webpage `http://httpd.apache.org/docs/2.2/mod/worker.html`, 2008.

[10] Bekman, S., and MacEachern, D. Installing mod_perl 2.0. Webpage `http://perl.apache.org/docs/2.0/user/install/install.html`, 2008.

[11] Burgess, M., and Begnum, K. M. Voluntary cooperation in pervasive computing services. In *LISA* [2], pp. 143–154.

[12] Burgess, M., and Couch, A. Modeling next generation configuration management tools. In *LISA* [3], pp. 131–147.

[13] Cons, L., and Poznanski, P. Pan: A high-level configuration language. In *LISA* (2002), USENIX, pp. 83–98.

[14] Couch, A. From x=1 to (setf x 1): what does configuration management mean? *;login: 33*, 1 (February 2008), 12–18.

[15] Couch, A., Hart, J., Idhaw, E. G., and Kallas, D. Seeking closure in an open world: A behavioral agent approach to configuration management. In *LISA* [1], pp. 125–148.

[16] Delaet, T., and Joosen, W. PoDIM: A language for high-level configuration management. In *LISA* (2007), USENIX, pp. 261–273.

[17] Delaet, T., and Joosen, W. Survey of configuration management tools. Tech. Rep. 485, K.U.Leuven, April 2007. `http://www.cs.kuleuven.be/publicaties/rapporten/cw/CW485.pdf`.

[18] Desai, N., Bradshaw, R., and Lueninghoener, C. Directing change using Bcfg2. In *LISA* [3], pp. 215–220.

[19] Fowler, M. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[20] Gelfond, M. On stratified autoepistemic theories. In *AAAI* (1987), pp. 207–211.

[21] Gelfond, M., and Lifschitz, V. The stable model semantics for logic programming. In *ICLP/SLP* (1988), pp. 1070–1080.

[22] Hess, A., Humm, B., Voss, M., and Engels, G. Structuring software cities a multidimensional approach. In *EDOC* (2007), IEEE Computer Society, pp. 122–129.

[23] Hewlett-Packard Development Company. *The SmartFrog Reference Manual: A guide to programming with the SmartFrog Framework*, March 2008.

[24] IBM. *IBM Dynamic Infrastructure for mySAP Business Suite: Operation and Administration*. IBM, 2007.

[25] IBM. Storage subsystems. Webpage `http://publib.boulder.ibm.com/infocenter/svcic/v3r1m0/index.jsp?topic=/com.ibm.svc.web.doc/svc_diskcontrollerovr_208hdd.html`, 2007.

[26] Jacob, B., Brokmann, M., Dickerson, S., Gomes, D., Hoppen, R., Lodhi, A., Madaan, K., Meels-Kurz, A., Oikawa, R., and Teixeira, T. S. *Deployment Guide Series: IBM Tivoli CCMDB Overview and Deployment Planning*. IBM International Technical Support Organization, 2008.

[27] Lankes, J., Matthes, F., and Wittenburg, A. Architekturbeschreibung von anwendungslandschaften: Softwarekartographie und IEEE Std 1471-2000. In *Software Engineering* (2005), P. Liggesmeyer, K. Pohl, and M. Goedicke, Eds., vol. 64 of *LNI*, GI, pp. 43–54.

[28] Madduri, H., Shi, S. S. B., Baker, R., Ayachitula, N., Shwartz, L., Surendra, M., Corley, C., Benantar, M., and Patel, S. A configuration management database architecture in support of IBM service management. *IBM Systems Journal 46*, 3 (2007), 441–458.

[29] Manoel, E., Brumfield, S. C., Converse, K., DuMont, M., Hand, L., Lilly, G., Moeller, M., Nemati, A., and Waisanen, A. Provisioning on demand: Introducing IBM Tivoli Intelligent ThinkDynamic Orchestrator. Tech. rep., IBM, 2003.

[30] Mercury. Optimize: Implementing ITIL change, configuration, and release management for maximum business value. Whitepaper `https://h10078.www1.hp.com/cda/hpms/display/main/hpms_content.jsp?zn=bto&cp=1-11%5E833_4000_100__`, 2006.

[31] Oppenheimer, D. The importance of understanding distributed system configuration. In *System Administrators are Users, Too: Designing Workspaces for Managing Internet-Scale Systems* (April 2003), Conference on Human Factors in Computing Systems.

[32] Oppenheimer, D., Ganapathi, A., and Patterson, D. Why do internet services fail, and what can be done about it, 2003. `http://roc.cs.berkeley.edu/papers/usits03.pdf`.

[33] Ortmeier, F. *Formale Sicherheitsanalyse*. PhD thesis, Universität Augsburg, July 2005.

[34] Randal, A. On topic: Topic and topicalisers in perl 6. Webpage `http://www.perl.com/pub/a/2002/10/30/topic.html`, October 2002.

[35] Rozenberg, G., Ehrig, H., Kreowski, H.-J., and Montanari, U., Eds. *Handbook of Graph Grammars and Computing by Graph Transformation*, vol. 3: Concurrency, Parallelism, and Distribution. World Scientific, 1999.

[36] samtregar. Re: Why aren't you using perl 6 yet? Webpage `http://www.perlmonks.net/?node_id=459783`, May 2005.

[37] Ward, C., Aggarwal, V., Buco, M. J., Olsson, E., and Weinberger, S. Integrated change and configuration management. *IBM Systems Journal 46*, 3 (2007), 459–478.

[38] Westerståndhl, D. Generalized quantifiers. In *The Stanford Encyclopedia of Philosophy*, E. N. Zalta, Ed. Winter 2005.

Appendix A

# READING PERL

Perl[1] is a dynamically typed programming language created by Larry Wall. It is strongly influenced by many of the common Unix shell utilities such as sed, awk, and the Bourne shell as well as more commonly used languages such as C, C++, and Lisp. One of the most defining characteristics of Perl is the ease of access to regular expression facilities in the language. The entire language, in fact, has a strong leaning toward strings. Since Perl has borrowed so heavily from commonly known languages this tutorial will try to focus much more on what might be seen as the things that cause Perl to look like "line noise" to many.

Perl is a very complicated language and so cannot be fully explained in a few, short pages. Perl also comes with very comprehensive documentation. To see a list of the manual pages that come with every distribution of Perl run:

```
perldoc perl
```

For more information on the language refer to either the "Programming Perl" or "Learning Perl" books.

In addition to the language itself, an important part of the Perl programming community is CPAN, the Comprehensive Perl Archive Network. CPAN is a repository of more the fifty thousand modules that can be easily installed and used. It has been so influential and important to the Perl community that it has been quipped that "CPAN is my programming language of choice, the rest is just syntax" [36].

### *Variables*

The type of a variable is determined by the sigil that precedes the identifier[2]. The main data types in Perl are

[1] Perl stands for the Practical Extraction and Report Language

[2] One can think of sigils as being like variable namespaces. They identify where to lookup a variable of a particular name. They therefore allow for `$var` and `@var` to be separate variables.

**scalar** Identified with `$` (dollar sign). Used for any single value data such as numbers, strings, and references.

**array** Identified with `@` (at sign). Used for holding a list of scalars.

**hash** Identified with `%` (percent sign). Used for holding a map from keys to scalar values.

Although by default Perl does not require variables to be declared it is common practice to use the `strict` pragma to make using undeclared variables a compiler error. To declare a variable a scope of the variable is provided (often `my`, or lexical, scope), followed by the variable sigil and name.

```
my $y;
my $x = 1;
my @a = (1, 2, 3);
my %h = (a => 1, b => 2);
```

Accessing the elements of arrays is similar to most languages, but instead of prefixing the array variable with `@` it needs to be prefixed with `$`[3]. Accessing a hash looks similar to accessing an array, but curly braces are used instead of square brackets.

```
my $element = $a[0];
my $other_elem = $h{a};
```

Working with data structures in Perl requires one to work with references instead of directly with hashes and arrays. Creating a reference to an already existing object (hash, array) is done by the \ (backslash) operator.

```
my $array_ref = \@a;
my $hash_ref = \%h;
```

Accessing a reference is similar to the syntax from C when one wants to access a pointer to a struct: an `->` is placed between the variable and the subscript.

```
my $val = $array_ref->[0];
my $other_val = $hash_ref->{a};
```

[3]This is because the sigil identifies what will come *out* of the variable expression.

One can also dereference a reference by prefixing it with the sigil of the type to dereference it into. In this form it is common practice to use curly braces around the reference to make it clear that a dereference is occuring.

```
my @a = @{ $array_ref };
my %h = %{ $hash_ref };
```

When working with an array the subscript starts at 0 just as in most other languages. However, one can also use negative subscripts to access the array offset from the end of the array. So the program

```
my @a = (1, 2, 3, 4, 5);
print "$a[-3]\n";
```

will print 3. This example also shows variable interpolation into strings. The string will be interpreted and the array access evaluated and substituted into the string before being printed.

There are two ways to discover the length of an array. The first is with the keyword `scalar` and the second is a more complicated sigil. The `scalar` keyword places the array into scalar context which provides the length of the array. The sigil `$#` will provide the index of the last element in the array.

```
my @a = (1, 2, 3);
print scalar(@a) . "\n";
print $#a . "\n";
```

The above program will print first a 3 (the number of elements in the array) and then a 2 (the index of the last element in the array). This example also shows string concatenation with the `.` (dot) operator. It also shows the ability of Perl to cast between types very easily. Although the array expressions returned numbers they were converted to strings to perform the concatenation.

Perl provides several operators to make a distinction between treating values as strings or as numbers. This table provides the equivalences.

| **Numeric** | **String** |
|---|---|
| == | eq |
| <= | le |
| >= | ge |
| < | lt |
| > | gt |

As stated earlier, when declaring a variable in Perl the scope of the variable is declared. The most common way of declaring a variable is with `my`, which creates a lexically scoped variable (what is most commonly seen in other languages). There is also `our`, which creates a package scoped variable, and `local` which dynamically scopes non-lexical variables.

A package scoped variable is placed the symbol table of the package in which it is declared and is therefore accessible from other packages. This differs from lexical variables which are only accessible from the same lexical scope as the variable declaration.

Dynamically scoped variables are changes to package scoped variables for the dynamically enclosing scope. These are not used often, so the exact mechanics and semantics are not important here.

### *Flow Control*

Perl uses many flow control constructs that are familiar to anyone familiar with other C family languages. However, in addition to normal `if() {} else {}` statement there is also a negated version: `unless() {}`. The `unless($x)` construction is equivalent to `if(!$x)`. A sometimes useful feature of `if` and `unless` is that they can be used as modifiers for a statement by placing them after an expression, in which case the meaning is the same as if the expression was the only element of the block of the conditional. This means that

```
if($x) {
        print "$x␣is␣true\n";
}
```

is the same as

```
print "$x␣is␣true\n" if $x;
```

Perl also supports C-like `while` and `for` loops. Additionally there is a `forach` loop to iterate over arrays. A foreach loop looks like

```
foreach my $x (@a) {
        print "$x\n";
}
```

which loops through the elements of `@a`, setting `$x` to each in turn and evaluating the block.

Drawing from the Lisp and Unix influence there are two more commands in the language for iterating over lists: `map` and `grep`. `map` applies a block to every element of a list and returns a newly constructed list of the outputs of the block, whereas `grep` takes a block and a list and returns a new list limited to the elements of the given list where the block evaluated to true.

### *Subroutines*

Perl supports both named and anonymous subroutines. To declare a named subroutine the keyword `sub` is given, the name of the subroutine, and then the body.

```
sub foo {
        print "inside␣foo\n";
}
```

Anonymous subroutines omit the name.

```
my $foo = sub {
        print "inside␣foo\n";
};
```

The anonymous subroutine provides a code reference and so can be placed in a scalar. It also creates a closure of its lexical environment[4].

Calling a named subroutine looks exactly like most other languages.

```
foo(1, "a", $y);
```

[4]Actually, the named subroutine does as well, which is important for `my` variables outside of subs and is used by many object systems in Perl.

Calling an unnamed subroutine requires the same reference syntax used by arrays and hashes.

```
$foo->(1, "a", $y);
```

Although `foo` was not declared with any parameters, these two examples are still perfectly legal. Perl does not declare formal parameters for subroutines and so does not check parameters passed[5]. To access the parameters that are passed to the subroutine each subroutine has access to the special variable `@_`. There are two common idioms for accessing the parameters of subroutines. The first looks like a declaration of formal parameters and relies on the ability of Perl to perform list assignments.

```
sub foo {
        my ($first, $second, $third, @rest) = @_;
        #...
}
```

The second form uses the `shift` operator to remove elements from the front of `@_`.

```
sub foo {
        my $first = shift;
        my $second = shift;
        my $third = shift;
        my @rest = @_;
        #...
}
```

The return value of a subroutine can be provided by using `return`. If execution reaches the end of the subroutine without encountering a `return`, then the value of the last expression is used. The return value of subroutines can be either a single scalar or a list. This allows subroutines in Perl to have multiple return values.

```
sub foo {
        my ($x) = @_;
```

[5]This is not strictly true. When a prototype is declared for a subroutine, then the number of parameters is checked. Prototypes are not commonly used, however. In fact many of the statements made in this section are not strictly true because of parts of the language that are not being explained. It is a philosophy of Perl, however, that what you do not know will not hurt you.

```
        if($x > 2) {
                return $x - 1;
        }

        return ($x + 1, "x␣was␣less␣or␣equal␣to␣2");
}
```

### *Objects*

The built-in object system of Perl is very primitive. It consists of mechanisms for creating objects, declaring inheritance, and method lookup, and not much more. Although the system is very simple, it is also very powerful when combined with the rest of the features of Perl. This section provides a quick introduction to the basic Perl object system and then a quick introduction to the Moose object system for Perl.

A class in Perl is a package. The `package` keyword declares the package in which the code following it should execute. A package in Perl is a symbol table in which to install `our` variables and named subroutines. To create a simple class then, one declares a package and the methods of the class.

```
package Animal::Dog;

sub wag {
        my ($self) = @_;
        # do something
}
```

To create an object there is no `new` keyword. Instead a reference is "blessed" into a package using the `bless` keyword. Blessing a reference creates a tag on the reference that tells the interpreter to start searching in the blessed package for methods. The most common reference to bless is a hash, since it can easily be used for storing instance variables. Usually the class also provides a `new` method that performs the bless, as well as any other initialization that is needed.

```
package Animal::Dog;
```

```
sub new {
        my ($class, $name) = @_;
        return bless { name => $name }, $class;
}

sub wag {
        my ($self) = @_;
        print "$self->{name} wags his tail\n";
}
```

This example allows an `Animal::Dog` object to be created and methods to be called on the object.

```
my $dog = Animal::Dog->new("Sam");
$dog->wag();
```

On top of these simple mechanisms have been built several object systems to make object oriented programming in Perl easier[6]. The program in this thesis uses Moose[7].

Moose provides a default `new` method for classes. Each class can implement a `BUILD` method to provide initialization code for the objects. Moose also allows declaration of attributes. The attribute creation can also automatically create accessor and mutator methods, type checking, default values, initialization from object construction, and many other useful features. Moose provides mechanisms for installing `before`, `after`, and `around` handlers for methods. It also has many meta-object facilities for supporting introspection of objects in the system.

```
package Animal::Dog;

use Moose;

has 'name' => (is => 'ro', required => 1, type => 'String');
```

[6]This is very similar to what happened in Lisp for building CLOS. In fact several of the Perl object systems have used ideas from CLOS for their design.

[7]Moose is influenced by CLOS as well as the Perl 6 object system.

```
sub wag {
        my ($self) = @_;
        print $self->name . "␣wags␣his␣tail\n";
}
```

***Miscellaneous Perl***

Although the basic information provided in the previous sections should be enough to understand most of what is used in the implementation of the language in this thesis, there are three other topics that need to be addressed: symbol table manipulation, exceptions, and truth.

The package `Coeus::Interpreter::AST` performs symbol table manipulation in order to generate code and make it callable as normal subroutines. As mentioned when talking about `our` variables and classes, a package in Perl creates a symbol table that can have variables and subroutines installed in it. One of the most common idioms for this is to place a new subroutine in the symbol table so that it can be accessed as a normal, named subroutine. The symbol table is accessed through what is called a "type glob," which can simply be assigned to.

```
*{ "foo" } = sub {
        print "inside␣foo\n";
};
```

Exceptions in Perl are done using the `die` keyword. An exception can be an object, a string, or even a subroutine. Any reference can be thrown with `die`. An exception is caught with `eval`. When the block evaluated by `eval` throws an exception, `eval` stops evaluating the block and stores the thrown exception in the `$@` variable. This variable can then be checked and any error handling performed.

```
eval {
        die "error!";
};
if($@) {
        print $@;
}
```

---

Truth in Perl shows its close association to strings and implicit conversion between types. False values in Perl are `0`, `"0"`, `""`, `undef`, and the empty array `()`. Anything not false is true. The negation of a true value provides a special false value that becomes the empty string `""` when used as a string and `0` when used as a number.