Practical Description of Configurations for Distributed Systems Management

by

Jim Thornton
B.Math (Computer Science) University of Waterloo, 1992

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENT FOR THE DEGREE OF MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES
DEPARTMENT OF COMPUTER SCIENCE

We accept this thesis as conforming
to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA
September 1994

Abstract

The administration of distributed systems is an important practical problem. One of the significant parts of the problem is the management of software configurations. The size and complexity of distributed systems have made automation of software management tasks essential. The time has come to determine how to design systems with intrinsic features that enable general management.

Experience with general approaches to software management is needed. This thesis presents a model that revolves around structured, declarative specifications of correct configurations. It is possible to use declarative specifications to automatically check the correctness of a system and also to automatically fix various problems. The model relies on an abstract view of systems as collections of objects with particular attribute values.

A new language is introduced for expressing configuration descriptions abstractly. Simple processing algorithms are given for automatically comparing a system with a description, and automatically eliminating discrepancies. A prototype implementation is described, and various related issues are explored.

The proposed model and language are suitable for practical use, as is demonstrated by an experiment involving a production system. While further work is needed in a variety of areas, the feasibility of using declarative specifications according to a general, abstract model has been established. This approach is not of merely theoretical interest. It can be applied to common systems in routine use today.

Table of Contents

List of Tables

List of Figures

Acknowledgment

This thesis owes a great deal to the excellent supervision I received. Gerald Neufeld managed a remarkable balance between allowing me to follow my own path, even when he didn't fully understand it, and keeping me on track towards a manageable thesis. I am grateful for the opportunity he gave me to pursue my own interests in a poorly defined area, and for his guidance that ensured the emergence of a well defined project in the end. Norm Hutchinson willingly gave of his time to listen to my ideas and make comments, even without any official obligation to do so.

I want to give special mention to Terry Coatta, who supplied inspiration, advice, and encouragement. Terry demonstrated an unusual level of comprehension of my ramblings, which was very encouraging. Our debates helped refine my ideas at critical points.

My family has encouraged me from the beginning, even though starting this program meant moving away to the other side of the continent. Thanks also to friends, both inside and outside the University, who have helped me in various ways throughout this experience.

Finally, thanks be to God, who made all this possible.

CHAPTER 1 Introduction

1.1 The Management Problem

Computer systems require administration. A large part of that administration is managing software configurations. Software is a key component of any modern computing system. Typically, software provides a great deal of flexibility and can be adapted to many situations. Flexibility contributes to the power or value of software as a tool, but it comes with a price. In any particular situation, there are many details which must be exactly right in order for software to function properly. Many people have experienced the frustration of trying to use a system that does not function as it should "merely" because it is not set up correctly.

Unfortunately, managing software configurations is no small task. In the first place, software is notoriously complex stuff. A great deal of specialized knowledge is often required to configure it correctly. Simplicity of administration is not always a priority when systems are designed. Configurations are also quite fragile, so apparently small adjustments can have large effects.

Dealing with continual change is also a challenge. Configuration changes must be made as people come and go, as new software is acquired and old software becomes obsolete, and as the computing needs of the organization change.

Managing the software for a single computer is enough of a challenge, but many organizations have computing infrastructures composed of hundreds of machines that function as a loose confederation. In these situations, it is not only necessary to configure each individual machine correctly, it is also essential to ensure that autonomous machines have compatible configurations. There is often a need to duplicate configurations as well, in order that multiple machines can be used for identical functions.

Large sites typically employ skilled, professional system administrators to acquire appropriate technology, keep everything running properly, and support users.With hundreds of machines to look after, however, the magnitude of the configuration task exceeds even the capacity of teams of professionals. Since the problem is the management of advanced computing systems, it is natural that we turn to automation and software tools for solutions.

1.2 Automating Configuration Management

Routine configuration management consists mostly of repetitive sequences of simple operations to configure software properly, to ensure that consistency is maintained, and to check that everything remains in proper order. These activities are ideal candidates for automation.

Successful use of automation has the potential to offer a number of clear benefits to end-users. System problems may be reduced by the elimination of some human errors, and the early detection of erroneous configurations generated by human actions or faulty software. Changes in configuration, such as the installation of new software, may be able to be completed more rapidly and more frequently, with less chance of error. More elaborate or varied configurations may be able to be supported within an organization. Ultimately, end-users may be able to safely perform more configuration operations themselves.

The most effective approach to automation for the long run is not to build more and more specific tools to solve particular problems as they are encountered. Instead, we should identify the general facilities and services that are required, and start building systems which include or support them. Work in this direction has begun, with the emergence of standards like SNMP [11] and CMIS/CMIP [12][13], and research developments such as RCMS [4]. This thesis takes another small step along that path, by proposing a general solution to some of the practical configuration problems associated with large numbers of workstations.

The thesis deals with a particular slice of configuration management, specifically, the problems associated with keeping large numbers of machines configured correctly. The emphasis is therefore placed on aspects of configuration that change infrequently, such as the values of server parameters, rather than details that change often during normal operation, such as the number of worker threads in a server process.

1.3 Existing Technology

A variety of tools have been developed to help automate different aspects of configuration management. This thesis addresses the *lack of generality* of many of those tools.

Many tools lack generality of application. They are frequently designed to deal with only some aspects of configurations (e.g. files and directories, but not filesystem definitions or processes). A tool may impose a particular software organization that is appropriate for some sites, but not for others.

Generality of function may also be lacking. A particular tool may be designed to either set up configurations (the most common case), or check configurations, but not both. Those designed to do setup frequently use procedural specifications (scripts), while checkers generally use declarative specifications (descriptions).

1.4 Thesis Statement

It is possible to automate configuration management operations of practical importance based on a general model that revolves around declarative specifications. Descriptions of correct system states may be used both to automatically check the validity of configurations, and to automatically generate the appropriate commands to bring the system in question into a correct state.

1.5 Thesis Contributions

The thesis makes the following significant research contributions:

1. A general model is defined for describing software configurations and automating operations based on such descriptions, in a practical context. The model emphasizes powerful, declarative, and structured specifications.

2. A new language for expressing descriptions of configurations is introduced. The Prescription language is declarative, yet at the same time it is designed so that simple, efficient algorithms can be used to discover a sequence of operations that may be performed to correct deviations from a specification.

3. A prototype implementation is described. In the course of producing the implementation, many issues significant to the practical use of the model and language were discovered.

4. An experimental application of the model to the configuration of real machines at the University of British Columbia is analyzed. The trials demonstrate the feasibility of the new approach and the new language.

1.6 Thesis Scope

In an area as broad as configuration management, it is not possible for a thesis such as this to address every issue. Unavoidably, there are significant omissions, some of which are mentioned here by way of advance warning.

The thesis is not directed towards the replacement of human administrators. In fact, the focus is on providing administrators with better technology so they can do a better job. The intelligence and creativity required by the administration task are unlikely to be emulated by software anytime soon. There may be ways in which general search and proof techniques from computational intelligence could be applied to extend this work, but that is outside the scope of the thesis.

Verification of the internal consistency of configuration descriptions is a very important topic. Constraints of space and time prevent any reasonable discussion of consistency checking.

The model is demonstrated by application to a realistic, but nonetheless limited, situation. Other areas of application need to be explored.

Dynamic, automated reconfiguration is not within the scope of the thesis, though it is consistent with the model. The task referred to here as dynamic reconfiguration is the adjustment of a system based on problems detected through monitoring or problem reporting. For instance, a routing table for an Internet router might be modified based on detection of the failure of a gateway. The emphasis here is on more static aspects of configuration.

1.7 Outline

The core of the thesis begins with the definition of the model in Chapter 2. The chapter begins with a more detailed analysis of the problem than was included in this introduction. Factors of size, change, complexity, diversity, repetition, and fragility are all considered briefly. A number of requirements for a solution are derived. The model itself is then presented, beginning with definitions.

The Prescription language is introduced in Chapter 3. All the pieces needed to realize the model are presented. The chapter concludes with a small but complete example demonstrating the application of the language and model to an imaginary situation.

The implementation and discoveries it engendered are described in Chapter 4. The issues and problems presented in Section 4.7 are not all specific to the particular implementation described in the chapter. Most are general, and relate to any implementation of the model.

The final major contribution of the thesis is the experiment and analysis presented in Chapter 5. Both the model and the implementation are involved in the experiment.

Other important matters are reserved to the later part of the thesis. Some related work is described briefly in Chapter 6. Conclusions and proposals for further exploration are presented in Chapter 7. An appendix provides the details of the experimental specification.

CHAPTER 2 A Model of Distributed Configuration Management

2.1 Problem Features

The complexity of software systems of all kinds has grown to the point where humans have difficulty coping with them. The problems of size and complexity are well recognized in the software engineering domain, and also arise in the context of distributed systems management.

This thesis deals with configuration management in distributed computing environments. In the past, a large user community was usually served by a single computer. It has now become common to find networks of independent machines providing the computing resources for organizations. Each machine typically has its own operating system, local disk, and configuration. These diverse collections of systems are managed by small groups of people, just like the isolated mainframes used to be.There are a number of features of distributed computing environments which contribute to the management problem.

Size

A distributed computing environment features a large number of items which must be properly configured. A single department may operate hundreds of machines. Each of those machines may have many files and directories which are local, and many references to resources on the network. Even if most systems are to be configured identically, they must still be manipulated individually. The size problem is the basic reason why automation is essential. It is simply not feasible for administrators to perform every configuration operation manually on each individual machine.

Change

Configurations change constantly. Some changes are planned and significant, such as the introduction of a new machine or a new shared filesystem. Others are unplanned and may even be undesirable. For instance, someone may manually change the configuration of a system in such a way that the system breaks. The fact of constant change has a few implications for management automation. Administrators do not have the luxury of getting everything set up correctly once, and then leaving it alone. Configuration management operations will be performed regularly. Also, there is more to the problem than mere setup. Administrators need to verify the configuration of systems.

Complexity

Distributed system configurations become very complicated since they involve so many details. A large amount of arcane knowledge and skill is often required to properly understand and adjust configurations. The knowledge that is required consists not only of knowledge about how technology works, but also familiarity with various rules and conventions which are site-specific. The impact of complexity is increased when important details are either undocumented or implicit, as is often the case.

Diversity

A number of types of diversity are significant to the problem of management. First, there is diversity of systems. A single site is likely to have machines from different vendors, running different (or at least variant) operating systems. The prevalence of system diversity implies that automated configuration tools ought to be general. There is also diversity of configurations within a typical site. This requires administrators to keep track of which machines are configured in which ways. Finally, there is diversity across sites. Each computing environment is a little different in organization, requirements and conventions. Site diversity implies that automated tools need to be flexible.

Repetition

Distributed software configurations are highly repetitive. Consider the situation of a group of Unix machines all accessing filesystems over a network. For each (machine, filesystem) pair, there are a number of configuration details that must be arranged correctly on the machine. Some of these details are common across all the pairs, however. Simple repetition also occurs when machines are configured identically in some respect.

Fragility

Software systems are notoriously fragile. Small deviations from a correct state can result in complete failure. This property is a problem in configuration as well as in programming, and it implies that every detail is important. In fact, configuration management is all about getting the details right.

2.2 Requirements

Based on the features just described, a number of objectives, or requirements, for any general approach to a solution can be identified. Some of the objectives are less than precise, due to the nature of the problem. The model presented later in this chapter is designed to meet these requirements:

1. Automation. A solution must support as much automation as possible.
2. Verification. A solution must support automated verification of the correctness of configurations. Verification addresses the problems of fragility and change management. Automation in this area requires some way to specify the correct configuration of a machine.
3. Declarative form. A configuration management system should be designed to use declarative specifications to the greatest extent possible. Declarative specifications are easier to understand and reason about than scripts or programs, when the objective is specification of states rather than computation. A declarative form will minimize complexity and simplify automated verification. A declarative form also partially addresses the problem of system diversity, since it avoids the difficulties of different execution environments on different machines.
4. Abstraction. A solution needs to be abstract in order to generalize to many different types of systems. Idiosyncracies of syntax that are particular to certain systems should be avoided. The use of abstraction can also serve to limit complexity.
5. Flexibility. Any configuration management system needs to be flexible in order to cope with the diversity of configurations and diversity of sites. A specification formalism should be powerful enough to describe any configuration state. In addition, it should not impose a particular system organization.
6. Structure. A model must support structure in configuration descriptions. Appropriate structure can minimize repetition, and help people deal with the complexity of diverse configurations.
7. Synchronism. An automated system needs to keep systems synchronized with changing specifications, to avoid the problems associated with change and fragility.

8. Transparency. A model must provide for specifications which are as explicit and self-documenting as possible. Transparency helps reduce complexity.

In addition to the requirements based on problem features, there is one final requirement that motivates the design of the model presented in this thesis:

9. Practicality. The solution must be appropriate for practical use in managing non-research distributed systems.

2.3 Definitions

A few terms are central to the new model.

A *managed system* is a collection of related entities whose configuration is managed with automation. The entities may be hardware or software, but software is the focus of the thesis. The entities that comprise a managed system are not restricted to a single machine. A managed system will be assumed to include a network, the machines on the network, and all of the software entities existing on all of the machines.

A *component* is any distinguishable piece of a managed system. For instance, files, directories, entries in tables, ports, sockets, and processes are all components of a managed system containing Unix machines. It is important to note that *component* always refers to an actual piece of a managed system, in contrast to the term *object*, which is defined next.

An *object* is an abstraction which represents or *models* a component. As an abstraction, an object presents certain features of the component it models, but not every feature. In particular, an object represents the configuration state of a component. Various aspects of the state of a component are represented by values of *attributes* of the modelling object. For instance, an object that models a file will have attributes for size, permissions, and so forth. The object will not model the operations that can be performed on a component (such as **open** in the case of a file). Any operations that are associated with an object are configuration management operations.

2.4 The New Model

A new approach to a practical configuration management system is presented here, based on the objectives listed in Section 2.2. At the centre of the new model is the idea that any configuration state of a managed system may be viewed as the union of the states of a set of objects representing components in the system[1]. A specification of a configuration consists of a declarative description of the states of a set of objects. Such specifications are the basis for all automated configuration management operations. The automated procedures of interest are comparison of specifications with actual managed systems, identification of discrepancies, determination of sequences of operations which may be performed to eliminate the discrepancies, and execution of the operations.

Through the use of objects, the model meets the abstraction and flexibility objectives of Section 2.2. Objects, as defined earlier, are abstractions that can be used to represent many different kinds of components. A specification language based on objects requires only one uniform syntax. Tools built according to this model are inherently flexible because the primary functionality is based on general abstractions, rather than the peculiarities of certain types of components.

It is not obvious that the new model can satisfy the conflicting requirements of automation, verification, and practicality. The majority of the thesis is devoted to showing that the use of declarative specifications does not preclude practical automation.

There are two parts to the model, each of which will be described separately. The specification part prescribes a particular structure for descriptions of configurations, and addresses the

1. This is the basic idea behind the general model introduced in [4].

requirements for declarative form, structure, and transparency. The automation part prescribes a particular approach to software tools, and addresses the requirement for synchronism.

2.4.1 Specification

Specifications of configurations (also called descriptions) have two parts. One part consists of a set of parameterized descriptions of object states, written in a simple logic language. The other part consists of a collection of data in a simple relational database. The database entries are referenced in the parameterized descriptions.

The logic language must be amenable to two different types of automated processing. The process of comparing a managed system with a specification is called *verification*, and a declarative language is naturally well suited to it. As mentioned earlier, however, automated procedures must also be able to generate, and execute, a sequence of operations to bring a managed system into conformance with a specification. This second process is called *repair.*

The database part of configuration descriptions is important for structural reasons. A logic language may supply *descriptive adequacy*, that is, it may permit any configuration state to be described. A language alone, however, does not provide enough structure to avoid repetition. The example of shared filesystems is helpful at this point. On each machine that imports a filesystem, a certain configuration must exist. Many details should be common across filesystems, but there are also details that are unique to each particular filesystem. For example, the mount point directory might be placed in /nfs and given the name of the filesystem, by convention. In that case, only the name of the required directory varies between filesystems. The name is an example of *instance data.* The mount point directory should be described only once in a specification, with the filesystem name as a parameter. A simple database table is a convenient form for the presentation and manipulation of instance data values.

Flexibility in this two-part specification form is achieved by leaving the definition of database schema and the creation of descriptions in the logic language up to administrators. The generic descriptions define the organizational structure and explicitly incorporate local rules and conventions. The database contains only the details which vary, separated from the repeated details.

The Prescription language, presented in Chapter 3, is designed to balance the need for declarative power with the requirement for practical repair. The language also supports modularity, to permit structuring and repetition avoidance, and in-line comments, to enhance transparency.

2.4.2 Automation

The basic principle of the model concerning automation is that all automation revolves around specifications. A variety of specific tools are possible. For example, a tool could be created to examine part of a managed system and produce a specification that describes the configuration.

In order to maintain synchronism of a distributed system with changing specifications, a configuration management system must be designed to operate continuously, and not just at infrequent intervals. For instance, adjustments to a machine configuration must not be delayed until the machine is next rebooted. The configuration management system must have regular access to machines in the managed system.

The majority of the software that performs automated processing based on specifications should be designed to operate on objects through standardized methods, in order to maintain the generality benefits of the abstraction. The implementation of the methods must manipulate real components, providing a translation between the object abstraction and the component reality. The issues associated with such a translation are extensively explored in Chapter 4.

CHAPTER 3 Managing Configurations with the Prescription Language

This chapter presents the Prescription language, which is designed to be used for describing software configurations for management automation, consistent with the model introduced in the previous chapter.

3.1 Introduction

The Prescription language is a tool for realizing the new model. According to that model, all managed components are abstractly represented as objects with attributes. The entire configuration state of a component is assumed to be represented by the values of attributes associated with an object. A specification consists of entries in some tables plus some logic descriptions of object states, in terms of the data in the tables. In order to meet the flexibility requirement of the model, there are actually three functions which the tool set must support:

1. Definition of object classes to model the various types of components
2. Definition of tables which will contain instance data

3. Description of object states

This chapter deals with each of the three functions separately, in order. The chapter concludes with a short example that combines all the pieces.

The heart of the Prescription language is the set of statements for describing object states. These statements involve only the object abstraction, since even the contents of tables are modelled by objects. The situation is not nearly as simple for table and class definitions. Table definitions must refer to specific files that will hold the data, and class definitions must be related to implementations which map the object abstraction to the component reality. For these functions, therefore, the details presented in this chapter will be partly dependent on the nature of the prototype implementation.

3.2 Syntax

The Prescription language could be implemented using any one of a myriad of syntax styles, from a Pascal style, to a C style, to a LISP style. A good syntax for the language would be clean, simple, and relatively terse. The ideal style would be one with those characteristics and some general appeal to the user community. The style used in this thesis is determined by the implementation.

Here are the main points which are important for understanding the examples:

1. A text consists of sequences of statements, one per line. The backslash (\) serves as a continuation character, and the semicolon (;) may be used to separate statements written on the same line.
2. Braces ({ and }) are used for grouping. For this reason, blocks of statements can be written to *appear* like blocks in C.
3. The string is the basic data type, and string literals do not need to be quoted unless they contain whitespace characters. Either double quotes (" ") or

braces may be used to quote strings[1]. The following two strings are identical:

```
"Hello World" {Hello World}
```

4. In order to use the value of a variable, the name of the variable is preceded by a dollar sign ($). This convention is similar to that used in BASIC.
5. Comments begin with an octothorpe (#) and end at the following line break.
6. The value returned by a function may be used in-line by enclosing the command in square brackets ([and]).
7. A global variable may be accessed within a local scope if the name of the variable is given as an argument to the **global** command, as illustrated below for the variable `Window`:

```
global Window
```

8. Assignment to a variable is performed by the **set** command. The following command assigns the value 14 to the variable `abra`.

```
set abra 14
```

3.3 Classes and Types

Every object is an instance of a particular object *class*, as in standard object-oriented practice. Object references may be assigned to variables, so the definition of a class also introduces a type of the same name as the class[2]. There are also fundamental types such as `Integer`.

Each class defines the object abstraction for a particular type of component. An associated implementation performs the mapping between the abstraction and the managed system. In the context of the Prescription language, a managed system is defined by the set of object classes that are defined. The descriptive statements, and the processing algorithms associated with

1. There two quoting forms differ as to the treatment of embedded variable references. The distinction is not important at this point.
2. The prototype implementation supports very little type checking. For this reason, types will not appear everywhere they should in the examples. A production implementation would benefit significantly from static type checking.

them, are not restricted to describing files and directories. They can be used with any kind of managed system, as long as the appropriate classes are defined and implemented. This generality is the benefit gained from the object abstraction.

The class structure can be described briefly using standard terminology. Single inheritance is supported, and multiple inheritance is unnecessary. Class methods are supported, but no need for class objects or meta-objects has been identified. Inheritance is not used for class method lookup. In order to support the algorithms for processing state descriptions, a class must export standard interfaces; additional interfaces may be defined by the creator for implementation use. The standard interfaces will be described later. There is a predefined base class called Object, with appropriate default implementations of many of the standard interfaces. Container classes are explicitly distinguished from non-container classes. The details of class structure, apart from those mentioned above, are not significant.

In addition to the types associated with object classes, there are also fundamental types, as has been mentioned. The `Integer`, `Real`, `String`, and `List` types do not have class definitions or implementations. The `Integer` and `Real` types support common operations on numeric values. The `String` type is used for regular character strings, supporting operations such as concatenation. The `List` type is a simple homogenous aggregate type. A `List` declaration must include the type of the elements in the list.

3.3.1 Definitional Statements

A class is defined by a statement with the following form[1]:

```
class name parent contents def-block
```

1. In these syntax templates, parameters for which actual values must be supplied are set in slanted characters.

The *name* parameter is the name of the class. The *parent* parameter is the name of the parent class in the inheritance hierarchy. The *contents* parameter is the type of items contained in instances of the class. For non-container classes, a null string must be supplied as the *contents* parameter. The *def-block* is a block containing the attribute and method definitions for the class.

Attributes are defined as follows:

```
attribute name type qualifiers
```

The *name* parameter is the name of the attribute, which must be used in attribute references. The *type* parameter specifies the type of the attribute value. The *qualifiers* parameter is used to identify important characteristics of the attribute. For instance, an attribute might be marked as `immutable`.

Methods can be defined by name and formal parameter list. Implementations must be provided for each defined method, but the mechanism for doing so is an implementation detail, and is therefore beyond the scope of the Prescription language. It is assumed that conventional per-class instance data will be supported for each object.

3.3.2 Object and Attribute Reference

Initially, object references are obtained from definition statements. Thus the object for a table that forms part of a specification is obtained from the `table` statement, as illustrated in the next section. Object references may be assigned to variables. An attribute reference consists of an object reference, a period, and an attribute name, following the syntax of structure-member reference in Pascal or C. For example, the following reference is to the `number` attribute of the object contained in variable `Q`:

```
$Q.number
```

Attributes may have values of a fundamental type, or may be object-valued. Due to the second possibility, multi-part attribute references may be generated, using the syntax of nested structure reference in Pascal:

```
$F.son.height
```

In this case, the value of the `son` attribute of the object referenced by `F` is an object having an attribute `height`[1].

3.3.3 Example

The following is an example of class definition. Two classes are defined. Sample method definitions are not provided, as they are beyond the scope of this chapter.

```
class Carton Object {} {
   attribute name String immutable
   attribute size Real {}
   attribute colour String {}
}

class Wagon Object Carton {
   attribute name String immutable
   attribute first Carton
   attribute owner String
}
```

The first class (`Carton`) inherits from Object and has three attributes: `name`, `size`, and `colour`. It is not a container class, since the null string is supplied as the *`contents`* parameter. The `name` is of type `String`, and is declared to be immutable. The `colour` is also of type `String`, but has no special declarations. The `size` is of type `Real`.

The `Wagon` class is a container class. Any instance of `Wagon` contains objects of class `Carton`. Note that `Wagon` also has an object-valued attribute (`first`, of type `Carton`).

1. Note that only one dollar sign is required; we do not write $($F.son).height. The dollar sign specifies *variable* dereference, not *object* dereference, which is always performed implicitly. This peculiarity of syntax is a product of the prototype implementation.

3.4 Table Definition

The database part of a configuration description consists of a set of tables which form a simple relational database. For each table in a database, a certain amount of information is required in order to permit it to be used in a specification. The necessary information is supplied by a definition statement of the following form:

```
table name filename field-delimiter fields-list key subtables-list
```

The *name* parameter supplies the name of the table. The `filename` is the name of the file which contains the records in the table, with fields delimited by the `field-delimiter` string. The *fields-list* is a list of field definitions, each consisting of a field name and a declared type. The *key* parameter identifies a primary key by listing the names of the fields which compose the key, in order. The *key* parameter may also be empty, in the case of a table with no primary key. The *subtables-list* identifies tables which are sub-tables, and includes information describing the relationships. A sub-table is a table which may contain multiple records associated with each record in the primary table. The sub-table information is not essential for configuration management, but indicates the structure of the database, and can be used by browsing tools.

Here is a sample table definition:

```
table MachineGroup mgroup.table | { \
   {name String} \
   {subgroups List MachineGroup} \
   {members List Machine} \
} {name} {}
```

The name of the table is `MachineGroup`. The file from which records will be read is `mgroup.table`. Fields in that file are declared to be delimited by vertical bar characters, as indicated by the third parameter. The table has three fields, called `name`, `subgroups`, and `members`. The `name` field is of type `String`. The other two fields are both of `List` types. In the case of subgroups, a List of MachineGroup, and in the case of members, a List of Machine. The file mgroup.table must contain strings representing every field value. When an field is of an object type, the string used to represent a value is a foreign key identifying a particular object. In this example, there are no sub-tables, so the empty string is supplied as the final parameter. The MachineGroup table defines a graph of groups (intended to be a tree in practice).

A table definition causes a number of things to be done implicitly. First, a class is defined for the table. The name of the class is the name of the table with the string "Table" appended. The new class inherits from a standard class called simply Table, and is a container class. The objects contained in an instance of a table class represent individual records in the table. A table statement, therefore, also causes the definition of a class for those contained objects, with an attribute corresponding to each field. The name of this new class is the same as the name of the table. A table statement also causes the creation of an object to represent the table (an instance of the new table class), and the creation of objects to represent each record in the table. The object representing the table is generally assigned to a variable. Thus the previous example definition would normally be written something like the following:

```
set MGroups [table MachineGroup mgroup.table | { \
   {name String} \
   {subgroups List MachineGroup} \
   {members List Machine} \
} {name} {} ]
```

As a result of this statement, a class named MachineGroup would be implicitly defined for objects representing records from the file mgroup.table. A class named Machine-GroupTable would also be implicitly defined as a container class containing objects of class MachineGroup. An instance of this second class would be created and its reference would be assigned to the variable MGroups. Finally, the contents of file mgroup.table would be read and objects created for each record (each an instance of class `MachineGroup`). The `table` statement plays the role of the **new** statement in other object-oriented environments.

When a field has an object type, the values actually stored in the table file for that field are foreign keys. The class created for records in a table automatically translates a foreign key into the appropriate object when the attribute for such a field is referenced. In some situations, the foreign key value itself is required, rather than the object. To accommodate these circumstances, a special variant syntax for attribute reference is provided. The prepending of an @ symbol to an attribute name specifies that the key value is required, *not* the object identified by the key value. This syntax is illustrated by the following example:

```
    $F.@son
```

As the tables are modelled by objects, the descriptive statements that will be introduced shortly may be used to express specifications of constraints on the tables. Since the tables are considered part of the specification, the prototype implementation does not support any automated repair of them. Repair could be supported, however.

The definition of the database schema and the creation of records in the database are both the province of the administrators responsible for the managed system. The format of record storage and the mechanisms used for editing records are beyond the scope of the specification language, and are implementation-specific. The prototype uses text files with simple structure, but a complete RDBMS is equally possible.

3.5 State Descriptions

This section covers the heart of the Prescription language, namely those statements and features which are provided for the description of object states. The descriptive statements that will be presented here are designed to be used for the structural parts of configuration specifications.

The descriptive statements are inherently declarative, each expressing a description rather than a directive. Each statement may be easily evaluated against a managed system at a particular point in time, to determine whether or not the described state holds. Each statement, therefore, has a *truth value* when considered relative to a particular managed system.

Although the language is declarative, there are also simple processing algorithms associated with each kind of descriptive statement. For each statement, there is a procedure for determining a truth value. In many cases, there is also a repair algorithm. A repair algorithm is a procedure to efficiently compute a sequence of operations which could be performed to modify the state of the managed system so that a statement would evaluate to True.

In order to facilitate state descriptions, the object model must be elaborated slightly. Some objects are *collections* which contain other objects. The list, or set, of contained objects is an attribute of the collection object, but it is an attribute of special significance. A number of the descriptive statements reference the contents of a collection object implicitly.

For practical specifications, facilities for general computation are important. General functions are used for the generation of values from other values (e.g. string concatenation, simple arithmetic). The set of general functions, and the mechanism for defining new ones are implementation dependent. The only restriction is that general functions must never manipulate the managed system. If they were to do so, the specifications would no longer be declarative, and implementation problems would arise.

The Prescription language is analogous to traditional programming languages in a number of ways. Statements may be grouped in block structures. Modularity is supported through named, parameterized blocks of statements like traditional procedures. Variables are used in exactly the same fashion as in common imperative languages supporting call-by-value.

3.5.1 Prescriptions

The unit of definition in the language is the *prescription.* A prescription is a named block of statements, with a list of named formal parameters which are used in the block as local variables. A prescription serves as a modular description of part of the configuration of a managed system.

A prescription plays a role similar to that of a predicate in a logic programming language like Prolog. If the formal parameters are bound to particular objects, the predicate has a truth value. There is not, however, any facility for automatic selection of values for unbound variables, as is provided in Prolog. In operational terms, a prescription plays the role of a procedure in a language like Pascal. In recognition of the special nature of prescriptions, they are said to be *activated*, rather than *called* as procedures.

Prescriptions are defined in a global scope in a fashion very similar to function definition in C. The following is an example of the definition of a prescription with an empty block as the body:

```
prescription FirstOne {a b c} {
}
```

In this example, the name of the new prescription is `FirstOne`, and there are three formal parameters, named `a`, `b`, and `c`. Prescriptions may be defined to be *narrowed* by the addition of the keyword `-narrow`[1] after the keyword `prescription`. Narrowing affects processing in a way that will be explained as soon as the processing modes have been described.

1. The hyphen is part of the keyword, and marks it as an option switch.

Prescriptions may be activated in a fashion similar to traditional procedure call. Actual parameter values must be supplied for each formal parameter. In the following example, the prescription `SecondOne` contains a single statement which is a recursive activation of the prescription.

```
prescription SecondOne {a b} {
   SecondOne $a $b
}
```

Note that prescription activation follows the syntax of statements, in which arguments are separated by spaces. While recursion is legal, this particular prescription is nonsensical from a logic point of view, and could not be processed without error.

3.5.2 Processing Modes

Prescriptions may be processed in two ways, described as separate processing *modes*. In *verify* mode, the sole objective of processing is to determine the truth value of a description at a point in time. In *repair* mode, an additional requirement is added; the managed system must be modified to conform to the description if possible.

The central principle of all Prescription processing is minimality. Evaluation of a prescription, or a statement, terminates as soon as the truth value has been determined. This is the same strategy used in the evaluation of expressions in C. Another consequence of minimality is that processing never results in modifications to the managed system in verify mode. In repair mode, modifications may be made, but the set of modifications is always minimal. If a prescription describes the existence of a particular file with certain permissions, and the file exists with different permissions, the only change made during repair mode processing will be a change to the permissions of the file.

The processing mode is normally preserved across prescription activations. Overriding of the processing mode is possible through narrowing, which is explained in the next section.

Not every descriptive statement is amenable to automated repair. Some statement forms do not include enough information to automatically select a repair algorithm. The reason such forms are tolerated in the language is that they have descriptive value. A statement for which there is a repair algorithm is designated as *repairable*. Note that repair may fail for a statement that is formally repairable, because the availability of an algorithm does not guarantee that the algorithm can be successfully applied. In verify mode, all statements can be processed, but in repair mode, processing of non-repairable statements might generate an error.

3.5.3 Narrowing

In various situations the full descriptive power of the language is required, and automated repair is either unexpected or undesirable. For example, a constraint may be expected to hold, with human notification required if it does not. It is desirable to be able to place such constraints in regular prescriptions that may be subject to repair processing. In these situations, the prescription author needs a way to selectively disable automated repair. Restriction of the processing mode to verify mode is called *narrowing*. A narrowed statement or prescription is always processed in verify mode, regardless of the mode in effect in the containing block. The restriction to verify mode is preserved through nesting and prescription activation.

There are two forms of narrowing. Individual statements may be narrowed with explicit syntax, as described in the next section. Alternatively, a prescription may be defined as narrowed, as described in Section 3.5.1, which causes every activation of that prescription to be narrowed automatically.

Although it is of practical importance, narrowing does not affect the logical meaning of statements at all. It merely affects repair processing.

3.5.4 Blocks

As in many programming languages, Prescription statements are grouped in blocks. Unlike other languages, the blocks themselves have logical semantics. There are three types of blocks: And blocks, Or blocks, and Narrow blocks.

Blocks are themselves legal statements, and so may be nested inside each other. There are different syntax possibilities for representing the distinction between block types. Different delimiters could be used for different types of blocks. In the syntax used here, braces delimit blocks of all types, and block statements are introduced by a keyword that identifies the type.

Statements are processed in order of occurrence, subject to the principle of minimality described in Section 3.5.2. The reason for order significance is to make processing easier to understand, and prescriptions simpler to write. A consequence is that implicit dependencies based on order may exist between statements.

And Blocks

An And block has the value True if and only if each contained statement has the value True. Thus an And block represents the logical AND of the statements in the block. The following example block always has the value True (the `true` and `false` statements always have the values True and False respectively):

```
and {
   true
   true
}
```

The following block has the value False in all cases:

```
and {
   true
   false
}
```

Both And and Or blocks are repairable. The body of a prescription is always an implicit And block. An And block may be empty, in which case it has the value True.

Or Blocks

An Or block has the value True if and only if there is at least one statement in the block which has the value True. Thus an Or block represents the logical OR of the contained statements. The following block has the value True in all cases:

```
or {
   false
   false
   true
   false
}
```

In accordance with the minimality principle, processing of an Or block terminates as soon as the truth value is determined. This pattern holds in both verify and repair modes. Repair mode processing involves two phases. First, the block is processed in verify mode, as though narrowed. If the block has the value False, implying that repair is required, each statement is then processed in turn in repair mode, until repair is successful for some statement. At that point, processing of the block terminates. Should repair fail for all statements, the block terminates with the value False.

Narrow Blocks

A Narrow block is used to specify narrowing. Narrowing is intended to be applied to single statements[1]. The following example block has the value False in all cases, but automated repair will never be attempted on any of the statements.

```
narrow { and {
   true
   true
```

1. For implementation reasons, a block is used. If multiple statements are included in the block it has the value of logical AND.

```
   false
}}
```

3.5.5 If Statements

An `if` statement is defined, for expressing configurations conditionally. The statement has the following form:

```
if { expression } then-block ?else-block?[1]
```

The *`expression`* part is just a standard boolean expression which may be composed of variable references, operators and functions. When the expression has the value True, the entire statement has the logical value of the *`then-block`*. Otherwise the statement has the logical value of the *`else-block`* if present, and the value True if not. In the following example, the `if` statement has the value True:

```
if { 1 == 1 } {
   true
} else {
   false
}
```

When an `if` statement is processed, the *`expression`* is evaluated first, then the appropriate block is processed. The `if` statement is repairable.

3.5.6 Logical Statements

A logical statement describes the state of a particular object by describing the value of one of the attributes of the object. A logical statement may or may not be repairable, depending upon the parts that are included. The statement has the following form:

```
logical ?attribute-reference? ?op? expression
```

1. In these syntax templates, optional parts are enclosed in question marks.

The `attribute-reference` parameter is a reference to an attribute of a particular object, through a variable. The *op* parameter is the comparison operator used to relate the attribute value to the value given by the *expression*. For example, the following logical statement states that the value of the `count` attribute of the object referenced by the variable `a` is 1:

```
logical $a.count == 1
```

The value of the logical is the value obtained by applying the *op* to the attribute value and *expression*. When the statement is processed, the *expression* is evaluated, the attribute value is obtained, and the appropriate comparison is performed. If repair is required, a suitable value is determined for assignment to the attribute. In some cases, the value to assign is trivially obtainable, as in the above example. Other operators make determination of a suitable repair value more difficult.

The `attribute-reference` and *op* parameters are optional. If they are omitted then the logical is non-repairable, because it does not include any identification of a particular piece of the managed system that can be modified. In the non-repairable case, the expression must evaluate to a logical value, which becomes the truth value of the entire statement. Here is the same constraint as in the previous example, expressed in non-repairable form:

```
logical {} {} {$a.count == 1}
```

Note that empty strings, delimited here by braces, are supplied for the first two parameters.

Logical Statement Operators

A wide variety of operators are possible, but only a few have been implemented in the prototype. Those that have been implemented are summarized here. Note that the presence of separate operators for testing numbers and strings is due to the limited typing system of the prototype implementation.

Table 1: Logical Statement Operators

Operator	Description
`==`	Numeric equality
`!=`	Numeric inequality
`<`	Numeric less-than
`>`	Numeric greater-than
`s=`	String equality
`s!=`	String inequality
`s<`	String less-than (lexicographic)
`s>`	String greater-than (lexicographic)
`contains`	Membership in collection
`eq`	Deep object equality

For these operators, there are repair algorithms that can be invoked when the value of an attribute causes the relation to be False. There may not be a repair algorithm for every imaginable operator. The statement can only support operators for which there is a possibility of repair.

3.5.7 Forall Statements

The `forall` statement describes the configuration of a set of objects. The form of the statement is as follows:

```
    forall var type ?-closure? collection ?constraint? block
```

The *`var`* parameter is the name of a variable, whose type is given by the *`type`* parameter. The *`collection`* parameter is an object which contains the objects the statement is about. The optional *`constraint`* is an expression in terms of the variable *`var`*, which identifies the particular objects from `collection` that the statement is about. The *`block`* contains statements that describe a configuration of each selected object, in terms of *`var`*. The *`block`* here is an implicit And block. The optional `-closure` flag specifies that the statement is about all objects in the closure of the *`collection`* under containment. Specification of the closure of a collection is useful for hierarchical structures such as Unix directory trees. In the following example, the statement describes a configuration in which the `count` attribute of each object in the collection identified by variable `col` has the value 1.

```
forall v Vooble $col {} {
   logical $v.count == 1
}
```

The value of a `forall` statement is True if and only if the value of the *body* is True for every object satisfying the *`constraint`* (if supplied) and contained in the *`collection`* (or its closure, if specified). The type of the variable (`Vooble` in this example) serves as an implicit constraint. A collection may contain objects of different specific classes, although they are all instances of the same base class. The variable type is used to constrain the statement to objects of a particular sub-class.

The `forall` statement is processed by successively binding the objects in the *`collection`* to the variable *`var`* (defined in the scope in which the statement resides). If a *`constraint`* is supplied, then it is evaluated for each object, and those for which it evaluates to False are skipped. Those objects which are not of the specified type are also skipped. If the closure of the *`collection`* is specified, then objects are retrieved from collection objects hierarchically contained in the specified *`collection`*. For each selected object, the body *`block`* is processed. If the *`block`* evaluates to False for some object, the truth value of the statement is known, so processing terminates (unless repair mode is in force and repair is successful). The `forall` statement is repairable.

Optimization using the Constraint

The `constraint` parameter primarily serves a semantic role, but may be used to increase efficiency. Semantically, the constraint is used to select particular objects to which the `block` is to apply. In the interest of efficiency, the presence of a selection criterion may permit the statement to be processed without iteration through all candidate objects. Instead of blindly iterating, it may be possible to narrow the search space or even retrieve an appropriate sub-collection, depending on the nature of the collection.

3.5.8 Require Statements

A `require` statement describes the configuration of an object which must exist in a collection. The form is as follows:

```
require var type ?id? ?-closure? collection block
```

The `var`, `type`, `collection`, and `-closure` parts are identical to the corresponding parts of `forall`. The optional `id` is a value which uniquely identifies an object from the `collection` as the one which the statement asserts must exist. It can be viewed as a key, and could have multiple parts. The nature of the identifying value is dependent on the `collection` in general, but when the closure is specified a particular form must be used. That form is a path through the hierarchy, with each component identifying an object in a collection at a different level. In the prototype implementation, such a path is expressed in the syntax of Unix pathnames. Thus the following example describes the configuration of a file referenced relative to the directory identified by variable `dir`:

```
require F File /sub/one -closure $dir {
   logical $F.perms == 0755
}
```

A `require` statement has the value True if and only if there is an object in the *`collection`* (or its closure, if specified), identified by *`id`* (if supplied), for which the *`block`* has the value True. In the absence of an *`id`* value, there may be multiple objects for which the *`block`* has the value True. As in `forall`, the *`block`* is an implicit And block.

The processing algorithm used with `require` depends on whether an *`id`* value is supplied. With an object identification, processing begins with a search for the identified object. If it does not exist, the statement has the value False. If the identified object does exist, then it is bound to the declared *`var`* and the *`body`* is processed. Without an object identification, the variable is successively bound to the objects in the collection, until one is found for which the *`block`* evaluates to True.

A `require` statement is repairable only if an identification is supplied. The identification indicates precisely which object the statement is describing, and so enables automatic determination of the problem that is the cause of a False value. The two possible problems are:

1. The required object is absent from the collection.
2. The required object is present but is not correctly configured.

In case of the first problem, the repair action is to create the object in the collection with default values for attributes unrelated to identification. The situation then reduces to the second problem, which is handled by simply processing the body in repair mode.

3.5.9 Disallow Statements

A `disallow` statement describes objects that must not exist in a collection. The form is as follows:

```
disallow var type ?-closure? collection ?constraint? block
```

The parts of the statement are all identical to the corresponding parts of the `forall` statement. The difference is in the meaning of the statement. For example, the following statement describes a configuration in which there are no objects in the collection identified by variable `col` with size 5:

```
disallow v Vooble $col {
   logical $v.size == 5
}
```

A `disallow` statement has the value True if and only if there is no object in the *`collection`* (or its closure, if specified) for which the *`block`* has the value True. Once again the *`block`* is an implicit And block.

Processing proceeds as with the `forall` statement, except that the truth value is known as soon as the first object is found for which the body is True, rather than False.

For repair, the same difficulty arises with `disallow` as with `require`; a repair strategy must be selected. For `disallow`, however, the same strategy is always used. Offending objects (those for which the body has the value True) are always destroyed. In a situation where destruction would be inappropriate, the statement should be narrowed and offending objects should be handled manually.

The meaning of "destruction" in the above description is flexible, because it is dependent on the implementations of object classes. For instance, destruction might be equivalent to Unix **unlink**, which may destroy only a reference to an item, and not the item itself.

3.6 Relationship to First Order Logic

The Prescription language is, in essence, a logic language. It is derived from first order logic. The differences are due either to the goal of supporting automated repair, or to practical considerations. The relationships between the various descriptive statements of the language and expressions of first order logic are described here. This section can be safely skipped if the theoretical details are not of interest.

3.6.1 Blocks

The relationship between Prescription blocks and simple logic constructs is very straightforward. An And block is equivalent to the conjunction of the contained statements, while an Or block is equivalent to the disjunction of the contained statements.

Notably, there is no Prescription equivalent of simple negation. This is due to the problem of automating repair. Negation is possible in expressions, and the effect of negation can sometimes be achieved by selection of the correct logical operator. Also, the `disallow` statement is related to negation.

3.6.2 Forall

The `forall` statement is equivalent to universal quantification in first order logic. The translation is quite straightforward. A statement of the following form:

```
    forall x XType $col { E(x) } { A(x) }
```

(where E(x) is an expression involving x and A(x) is a statement involving x) is logically equivalent to the following expression in first order logic:

$$\forall\ x \in \$col,\ (\ x \propto XType \wedge (E(x))\ ) \Rightarrow (A'(x))$$

where $\alpha \propto \beta$ means that α is type conformant to type β, and X' indicates the expression in first order logic which is equivalent to the prescription statement X. Similarly, a statement of the following form:

```
    forall x XType -closure $col { E(x) } { A(x) }
```

is equivalent to the following expression:

$$\forall\ x \in \mathbf{closure}(\$col),\ (\ x \propto XType \wedge (E(x))\ ) \Rightarrow (A'(x))$$

where **`closure(`**φ**`)`** is the closure of φ under containment.

Note that the `forall` statement is structured to explicitly separate the left side of the implication from the right side.

3.6.3 Require

The `require` statement is equivalent to existential quantification in first order logic. A statement of the form

```
require x XType id $col { A(x) }
```

is logically equivalent to

$$\exists\ x \in \$col\ \mid (\ x \propto XType\ ) \wedge (x \sim id) \wedge A'(x)$$

where $\alpha \sim \beta$ means that α is uniquely identified by β. Similarly, the closure form

```
require x XType id -closure $col { A(x) }
```

is equivalent to

$$\exists\ x \in \mathbf{closure}(\$col)\ \mid (\ x \propto XType\ ) \wedge (x \sim id) \wedge A'(x)$$

Note that the `require` statement distinguishes particular parts of the expression (the type and identity constraints). The distinguished parts provide enough information to enable automatic selection of the correct repair strategy. The statement may be written without an identifying value:

```
require x XType {} $col { A(x) }
```

which is logically equivalent to

$$\exists\ x \in \$col \mid (x \propto XType) \wedge A'(x)$$

3.6.4 Disallow

The `disallow` statement is equivalent to negation of existential quantification. Therefore, a statement of the form

```
disallow x XType $col { E(x) } { A(x) }
```

is logically equivalent to

$$\neg\exists\ x \in \$col \mid (x \propto XType) \wedge E(x) \wedge A'(x)$$

The closure form

```
disallow x XType -closure $col { E(x) } { A(x) }
```

is equivalent to

$$\neg\exists\ x \in \mathbf{closure}(\$col) \mid (x \propto XType) \wedge E(x) \wedge A'(x)$$

3.6.5 Logical

The relationship between a `logical` statement and an expression in first order logic is very simple. A logical of the following form

```
logical $a.b.c Δ v
```

(where Δ is any operator) is logically equivalent to

```
$a.b.c Δ v
```

In the non-repairable form, the statement is simply equivalent to the expression part.

3.6.6 If

There is also a very simple relationship between an `if` statement and an expression in first order logic. A statement in the following form

```
    if { E } { A }
```

is logically equivalent to the following expression

$$E \Rightarrow A'$$

3.6.7 Derivation

The statements of the Prescription language were actually derived based on first order logic expressions. Each statement can be considered to express a useful idiom of logic in a manner intended to facilitate specifying configurations and performing automated repair.

3.7 Support Functions

As indicated in Section 3.5, general computation functions are required. Some of the significant functions implemented in the prototype are described briefly here.

3.7.1 Membership Testing

It is frequently necessary to test a collection for the presence of a particular object. For this purpose, the boolean-valued function `in` is provided:

```
    in ?-key? val collection
```

There are two forms of membership testing. Without the `-key` switch, a simple test for the presence of the object *`val`* in the *`collection`* is performed. When the `-key` switch is supplied, the *`val`* parameter is taken to be an object identifier like that used in the `require` statement. The function then returns True if and only if there is an object in *`collection`* identified by *`val`*.

3.7.2 String Concatenation

The most common value manipulation is the production of string values by concatenation. In many implementations, string concatenation would require an explicit function. In the prototype implementation, no special function is required. For example, the following string is the concatenation of the value of variable `a`, a stroke character (`/`), and the value of variable `b`:

```
$a/$b
```

3.7.3 Special Purpose

A number of special functions are used in the prototype implementation. In another implementation, operators might handle these cases. The functions are summarized in the following table:

Table 2: Miscellaneous Functions

Function Form	Description
`val` *`aref`*	Returns the value of an object attribute. The desired value is identified by an attribute reference *`aref`*. This function is only required in certain situations, due to the nature of the implementation.
`objEQ` *`a b`*	Boolean function which tests two objects (referenced by *a* and *b*) for identity.
`globEQ` *`pattern b`*	Function to test a string *b* for a glob match against the *`pattern`*
`strEQ` *`a b`*	Convenience function to test two strings (*a* and *b*) for equality
`strNE` *`a b`*	Convenience function to test two strings (*a* and *b*) for inequality

3.8 Repair Limitations

The Prescription language has considerable descriptive power. Despite the fact that it is designed to support automated repair, there are many configuration descriptions that can be written for which repair will not be successful. Each major barrier to repair is presented briefly here.

3.8.1 Explicit Narrowing

As described earlier, it is possible to write pieces of description for which repair is inhibited. This may be done whether or not the narrowed statements are repairable.

3.8.2 Implicit Narrowing

When a non-repairable statement is encountered during processing, it may be implicitly narrowed. This is the case with the `require` statement when the key value is omitted.

3.8.3 Solver Inadequacy

Repair always proceeds in a simple fashion without backtracking. As a result, there are specifications for which repair will fail to produce a correct result, even though there is a perfectly adequate solution. Consider, for example, the following block:

```
and {
   logical $B.b < "2 * [val $C.c]"
   logical $C.c < "[val $D.d] - 5"
}
```

Suppose that the block is processed in repair mode when B.b = 25, C.c = 10, and D.d = 10. The first logical statement will be determined to have the value False, since 25 is not less than 2*10. The repair action might be the assignment B.b $\leftarrow$ 19. The second logical statement would then be processed and determined to have the value False, since 10 is not less than 5. The repair action might be C.c $\leftarrow$ 4. While that action would cause the second statement to have the value

True, it would simultaneously cause the first statement to have the value False. The repair process would not detect this problem.

In considering this example, note first that there is an obvious solution which would cause both statements to have the value True simultaneously (B.b $\leftarrow$ 7, C.c $\leftarrow$ 4). Secondly, observe that in this particular case, simply reversing the order of the statements in the block would make it possible for the repair algorithm to compute a correct solution.

The problem which this example reveals is the inadequacy of the simple, naive repair algorithm when dealing with specifications involving certain dependencies. An obvious solution would be to use a more general solver, one capable of handling such dependencies and finding a solution if one exists. There are a few reasons why effort was not expended in this direction:

1. Awkward situations like that illustrated in the example do not seem likely to arise often in practice. The common situations in static system configuration are very simple.
2. Inclusion of a more powerful solver would greatly increase the complexity of an implementation, and could make its operation more difficult for administrators to understand.
3. A simple repair algorithm has efficiency advantages. It is possible (at least in theory) to write a specification equivalent in form to 3SAT, for which finding a solution is an NP-complete problem. Given the previous points, avoidance of an exponential algorithm is a good idea.

3.8.4 Object Limitations

Direct modification of certain attributes of certain components is not possible. For example, consider the `size` attribute of a file, which we might define to be the size of the file in bytes. Now suppose that the following description is written:

```
logical $F.size == 23
```

If the size of file F is 23 bytes, all is well. If not, the repair algorithm will easily determine that the appropriate repair action is the assignment F.size ← 23. The problem is that performing that repair action is non-trivial. For this example, a simple algorithm can be devised, since there are steps that can be taken to change the size of a file to any particular value. Unfortunately, it seems unlikely that an automated algorithm will produce a desirable result in most cases.

Repair will ultimately fail if it involves changing the value of attributes which are not directly modifiable. The `size` attribute in the above example is a *dependent* attribute; its value depends on other properties of the object. Other attributes might be *immutable*, that is, not subject to modification for some intrinsic reason. Using static analysis of specifications, an implementation could warn the user of possible problems with non-modifiable attributes.

3.8.5 Failures

Repair can also fail for unpredictable reasons. A repair operation may simply not work when it is attempted. For example, an attempt to create a file will fail if the disk space is exhausted. Without a more elaborate reasoning system, the only solution to such a problem is to report it to a human.

3.9 Repair Processing Details

A few practical details concerning repair remain to be discussed. The target application for the language is configuration of production systems. Administrators are only too aware of the problems that result when a system is inadvertently placed in an inconsistent state. Configuration management tools must be carefully implemented to avoid contributing to the very problems they are designed to solve. They must also be designed to satisfy the need of administrators to understand exactly what is being done on their behalf.

It is important that systems not be left in inconsistent states, as might occur if repair were to fail part way through processing a description. To avoid this situation, prescriptions should be handled atomically. Either a prescription should be True after processing, or the managed system should be in its original state. There are some problems with implementing atomicity as described. These problems will receive more attention later in the thesis.

The description of processing revolved around the two modes: verify and repair. In practice, verify mode is of limited utility, since it causes processing to terminate with the first encountered discrepancy. The difficulty with repair mode is that it involves modifications to the managed system which an administrator may need to specifically review and approve. The simple solution to these problems is to split repair processing into two phases. In the first phase, the repair actions to be performed are determined, during processing of the prescriptions. The actions are recorded in a log. Execution is deferred to the second phase, after processing of statements is completed. The administrator can review the log between the two phases. Tools could even be provided to permit manual editing of a log. Unlike the case with verify mode, the log can include a description of every way in which the managed system fails to match the specification.

The two-phase repair processing approach will be referred to as `deferred execution`. In addition to the advantages described above, it helps achieve atomicity, since statement processing finishes completely before the first modification is made. Use of deferred execution causes some implementation difficulties, however. It requires maintenance of a large amount of state information during the first phase, and special handling to avoid problems with dependencies on implicit side-effects. These problems are explored later in the thesis. If an implementation were to be used on an unattended basis, it might not be necessary to implement deferred execution, since atomicity can be maintained through a roll-back mechanism.

3.10 Standard Class Interfaces

As previously mentioned, class implementations serve to provide the translation between the uniform object abstraction used by the processing algorithms and the concrete syntax required to actually manipulate components. The processing code remains independent of the idiosyncracies of particular components (and, indeed, of particular operating systems).

The processing algorithms function by invoking certain methods on objects. Any component of a managed system can be handled as long as the class for that type of component supports the methods which are used by the processing algorithms. These methods are invoked during both verification and repair processing.

The standard interfaces may be organized into several groups by purpose. The following table describes the methods in each group. Class methods are indicated by "(class)" following the method name. Every class that is defined must support each of these methods. Default implementations are inherited from Object[1].

Table 3: Standard Object Methods by Group

Group	Method.	Description
Object Management	constructor	Initialize object
Attributes	getAttribute	Get the value of an attribute.
	getDefault	Get the default value of an attribute for a newly created component.
	setAttribute	Set the value of an attribute.

1. Collection methods are not required on non-container classes. The default implementations in Object simply raise an error.

Table 3: Standard Object Methods by Group (Continued)

Group	Method.	Description
Classification	claim (class)	Determine whether the object is an instance of a specific class, when it is known to be an instance of a general class.
Testing	isDeepEqual	Test an object for equality with another object using class-specific evaluation.
Collection Related	getCollection (class)	Obtain the object which holds all instances of a class, for classes whose instances are all generally held in a single collection object.
	initIteration	Begin iteration over objects in a collection.
	getNext	Retrieve the next object in the iteration.
	endIteration	Cleanup from iteration over objects in a collection.
	find	Locate an object by identifying value.
	prepare	Prepare a new object for addition to a collection.
	create	Actually create a new object (and component).
	destroy	Destroy an existing object (and component).
	ismember	Check for presence of an object in a collection.
Modification	commit	Commit changes to object.
Debugging	printRef	Produce human-readable version of object reference.

As an example, assume that we are interested in managing disk files. We need to define a class, say File, to model disk files. In this situation, the actual files are the components. Every instance of the File class is an object representing a specific disk file. We must provide implementations of the standard methods to manipulate the real files. For example, when **getAttribute** is invoked on a File object for the `size` attribute, the implementation must perform the operating system function to retrieve the size of the file represented by the object. For the Unix operating system, the implementation would use the **stat** system call. On another operating system, the implementation would use whatever function that system provides for obtaining the size of a file. The processing of a statement such as the following (with F referencing an instance of class File) would include an invocation of **getAttribute**.

```
logical $F.size == 150
```

Should repair be required for this statement, the **setAttribute** method would be invoked. For log files, we might define a LogFile class as a subclass of File. We could define the `size` attribute for class LogFile to be the number of lines in the file, then the **setAttribute** implementation could truncate or expand a file as necessary to reach any specified size.

Every attribute exposes some aspect of the configuration state of components modelled by the class with which the attribute is associated. A class implementor can define attributes in whatever way seems appropriate, and provide implementations to get and set attribute values. The processing algorithms operate using the standard methods, and incorporate no knowledge of particular components or operating systems.

We could model directories by defining a Directory class[1]. Since directories contain files, the Directory class would be a container class for objects of class File. The collection methods would be implemented on class Directory to operate on files in the directory represented by a Directory object. For instance, the **find** method would take a file name as an identifying value,

1. For Unix, at least, Directory would be a subclass of File.

and search the directory for a file with that name. Upon finding a file with the right name, the method would create an object to represent that file, and return the object.

The relationship of many of these standard methods to prescription processing is straightforward. There are a few methods whose purpose is not clearly explained by the above table, however. Some of these will be dealt with in the next section, but the rest can be explained immediately.

The **claim** method is required to deal with class inheritance hierarchies that model classification hierarchies. The prime example of this is Unix files. In Unix, a directory can be viewed as a type of file, as can a symbolic link, a device special file, and so forth. We can model this by defining a root class File, with child classes Directory, SymLink, etc. A Directory would then be formally defined to contain objects of class File. Suppose that we then want to write a prescription statement such as the following:

```
forall S SymLink $dir {} {
   logical $S.mode == 0755
}
```

When the statement is processed, objects are extracted from `dir`, but those objects may be declared to be of class File or Directory in addition to SymLink. The **claim** interface is used to determine whether the object is actually of the required SymLink class.

The **isDeepEqual** method is required to deal with object comparisons that cannot be made without knowledge of the particulars of the component. It is used to implement the `eq` operator of the `logical` statement.

The **getCollection** method is particularly unusual. It is used to translate foreign keys in the fields of specification tables into the objects they reference. This method is unlikely to be needed for modelling managed components.

3.10.1 Implementing Deferred Execution

Supporting the deferred execution strategy, which was introduced in Section 3.9, is a significant issue for the class implementor. There are two processing phases, and all modifications to managed components are deferred until the second phase. This causes a problem of tracking managed system state.

Deferring operations requires that the software keep track of the *intended* state of components that are slated for modification. To understand why this is so, consider the following statement:

```
require F File /tmp/example $root {
   logical $F.perms == 0755
}
```

Suppose that when the above statement is processed, the software determines that file `/tmp/example` does not exist and must be created. Using deferred execution, the creation will be logged, but not performed immediately. Processing will continue with the block in the `require`. Now the system must process the `logical` correctly, despite the fact that it refers to a file which does not exist (yet).

The problem here is the same one that occurs when one attempts to build a virtual file system that can diverge temporarily from the underlying stable storage. The solution is the same as well: the software must maintain enough information to process requests from memory, without referencing the underlying system directly.

In the context of implementing the Prescription language, there are two ways to organize the maintenance of state representation:

1. Object state can be maintained entirely by class implementations using object instance data.
2. Object state changes can be recorded centrally by the main processing code,

without involvement of the class implementations.

The second of these alternatives is attractive from the point of view of simplifying the class implementor's task, and reducing functional duplication between class implementations. For these reasons, the second alternative is supported in the prototype implementation.

The processing algorithms do not invoke methods directly. Instead, they call special procedures which may or may not invoke the appropriate method on an object, depending on the recorded state of the object. Only a few pieces of state information are maintained. There is a general state indicator, which records whether the component is pre-existing, slated for creation, slated for destruction, or slated for modification. Lists of objects to be added or deleted are maintained, to track changes to the membership of collections. Finally, the new value of any modified attribute is recorded.

By relying upon central record keeping, a class implementor does not need to worry about the possibility that an attribute value will have been slated to change. The class implementation can just access the modelled component to respond to every **getAttribute** request. If the attribute value does need to change, the new value will be centrally recorded, and subsequent requests for the value of the attribute will be satisfied without invocation of **getAttribute** on the object at all.

It turns out that there are a couple of situations in which this strategy fails. These account for two of the surprising methods in the standard set. First, when an object is created (through processing of a `require` statement), it is not possible for the central record to satisfy requests for the value of attributes, but it is also not appropriate to invoke **getAttribute**, because the managed component does not yet exist. The **getDefault** method is invoked instead of **getAttribute** in this case to retrieve the value that the attribute should have when creation actually completes. The second situation concerns creation itself. It is important that the class implementation for a collection perform the creation of any object to be added to the collection. The **prepare** method handles this part of creation.

The complete mechanism described above works well for some classes, but it does not solve every problem. One issue that remains is the handling of components that cannot be individually addressed in the managed system. For instance, entries in a simple Unix configuration file can be modelled by objects. The various fields in an entry can be treated as attributes. A problem arises in the second phase of repair, when changes are actually being made. The problem occurs when it is not possible to independently modify the field of one entry without affecting surrounding fields or records. In order to implement **setAttribute** under the scheme described above, it would be necessary to read the entire file and write it out again, changing the value of one field in the process. Now imagine what happens when a number of modifications are made to different records in the same file. The implementation just described would result in the file being read and written several times. With components like records in a file, which cannot be independently modified, the best implementation strategy is to make all modifications to a representation in memory, then simply regenerate the file after all the changes have been recorded. This approach requires the class implementation to track state changes.

Fortunately, the class implementation can do whatever is appropriate with the instance data, so there is no fundamental conflict with the mechanisms described previously. There is a need for a method to be called after all modifications from the log have been completed. The standard **commit** method is the one designated to fill that role. A list of every modified object is maintained. After processing of the log is finished, the **commit** method is invoked on each object in the list. A interface is supported to permit the implementation of a class to add an object to the list of modified objects. This is used, for example, so that the implementation of a class modelling records in a file can add the object modelling the file itself to the change list.

In summary, there are two implementation strategies available to the class implementor. Following the first strategy, the methods all reference the modelled component directly (except for those like **getDefault**), and state tracking is ignored. Following the second strategy, the class implementation maintains a complete model of the component state, and only references the component directly at construction time and during commit.

The mechanisms introduced here *still* do not address every problem. Attributes with inter-dependencies cannot be properly supported. For example, if a change to the value of one attribute implies a change to the value of another attribute, the situation cannot be modelled with the available mechanisms. A central record will be made of a modification to one attribute *without* any invocation on the object. To deal with inter-dependencies, more standard methods would be needed.

3.11 Example

The example presented here is intended to illustrate how all the elements described in this chapter fit together. A simple, hypothetical situation is used in order to limit the length and complexity of the example.

Consider the management of an electronic mail system. In this system, assume that there is a folder for each user. Folders may be created in a structure called a file cabinet. This managed system is modelled by the following classes:

```
class Message Object {} {
   attribute from String immutable
   attribute date Date
   attribute text String
}

class Folder Object Message {
   attribute name String immutable
   attribute group String
   attribute size Integer
}

class FileCabinet Object Folder {
   attribute name String immutable
}

class System Object FileCabinet {}
```

From these definitions, we note that the configuration state of a folder consists of a name, a group, and a size (the number of messages which the folder can contain), plus the set of contained messages. The configuration state of a file cabinet consists of a name and a set of contained folders. The method definitions are omitted here, because they are essentially implementation details.

For this managed system, assume the administrators wish to specify the details of the folders needed by students enrolled in various courses. All student folders will have the same size, and the administrators naturally want to avoid entering that value many times. The following table is adequate to hold the varying instance data:

```
table StudentFolder students.table | { \
   {name String} \
   {course String} \
} {name} {}
```

Note that there are two fields in every record. The name field holds the student's name, and serves as the primary key. The course field holds the designation for the course in which the student is enrolled (assuming that students are only enrolled in one course at a time, for this example). Here are some sample records:

Table 4: StudentFolders

name	course
John Tan	CPSC124
Mary Alder	CPSC128
Lucy Denter	CPSC126

Table 4: StudentFolders

name	course
Tom Roe	CPSC319
Jo Terry	CPSC415

Now a prescription is required to describe the configurations of objects in the managed system based on the data in the table. The following simple prescription is adequate in this case:

```
prescription StudentFolders {T} {
   require FC FileCabinet Students $System {
      forall E StudentFolder $T {} {
         require F Folder $E.name $FC {
            logical $F.group s= $E.course
            logical $F.size == 100
         }
      }
   }
}
```

The prescription describes the existence of a file cabinet with the name “Students” containing one folder for each record in the StudentFolder table. The name which identifies each folder is the name from the record in the table. The group for each folder is the course designation from the table record. Finally, the size of each folder is specified to be 100. The table must be supplied as the parameter `T` when the prescription is activated. The variable `System` is assumed to be bound globally to an object representing the entire managed system.

This simple example demonstrates how a managed system can be defined by a set of class definitions, how a table can be defined to contain the varying details of a specification, and how a prescription can be written to complete the specification and express the details which do not vary. The prescription above can be activated in repair mode to automatically configure the managed system, or to obtain a complete report of the ways in which the managed system deviates from the specification.

CHAPTER 4
Implementation

Implementation of the Prescription language poses a number of challenges. This chapter describes the details of the prototype, and presents the issues which are significant for any implementation.

The prototype described here was produced in order to explore implementation issues, and permit the experiment presented in the next chapter to be conducted. The implementation was never intended to be suitable for production use. It is "one to throw away."

4.1 The Environment

Unix is a natural choice for the prototype operating system environment. The examples considered during the development of the model and language were Unix related, and Unix is the primary computing environment in the Computer Science department at the University of British

Columbia. Despite the focus on Unix, both the language and the central ideas of the implementation are transferrable to other contexts.

For convenience, the prototype implementation was produced for SunOS 4 systems only. Most of the code should work on other systems unmodified. The sample class implementations, however, are quite specific to the operating system variant.

The prototype implementation is written exclusively in the interpreted language Tcl [1], and relies on the standard interpreter. There are a few reasons why Tcl is appropriate for this project.

1. Tcl provides facilities which permit extension of the interpreter to support new language features, including new control structures. It is important to realize that this involves more than a facility for defining new commands or procedures. Syntactically, the extensions appear as though they were part of the base language, which is particularly convenient.
2. Tcl has the basic features that are desirable for a prototyping activity. It is a very high-level language, supporting simple manipulation of strings and lists. It is interpreted, and it supports automatic storage management.
3. Through common extensions, Tcl has support for direct interaction with Unix and high-level RPC for writing distributed applications.

Since the Tcl interpreter plays a significant role in the implementation, the syntax has to conform to that accepted by the interpreter. Thus, it is actually Tcl syntax which has been used up to this point[1]. The Tcl syntax is acceptable, if less than desirable. The more serious disadvantages to the use of Tcl have to do with efficiency and the lack of typing. These problems will be explained later. Despite its limitations, Tcl serves the purpose admirably.

1. The origin of the syntax was not explained in the previous chapter to avoid confusion between the Prescription language and Tcl.

4.2 Basic Processing

The implementation relies on the Tcl interpreter, as already noted. A Tcl procedure is defined in the interpreter to implement each statement in the Prescription language. The prescriptions are actually executed by the Tcl interpreter.

The basic procedures that are defined to implement the descriptive statements of the language are `and`, `or`, `prescription`, `narrow`, `logical`, `forall`, `disallow`, and `require`. In addition to these, an `activate` procedure is defined to initiate processing, various support procedures (documented earlier) are defined, and the definitional procedures related to class and table definition are provided. A large number of other procedures are part of the implementation internals.

The blocks that are part of most descriptive statements are merely string arguments to procedures, as far as the Tcl interpreter is concerned (recall that braces are string delimiters). This is exactly how the basic control structures of the Tcl language itself are implemented. Braces inhibit substitutions. Thus when a statement such as:

```
frodo {
   puts "Hi there $mom"
}
```

is executed, the interpreter passes the contents of the braces as an argument to `frodo`, *without* performing a substitution of the value of variable `mom`. The implementation of `frodo` can invoke the interpreter on the argument, in the variable scope in which the call to `frodo` was made. At that point, the interpreter will perform the variable substitution.

In order to implement the Prescription statements in an interpreter, unusual transfer of the flow of control must be possible. Tcl provides the ability to throw and catch exceptions, which makes implementation of new control structures possible. The prescription implementation uses three special exceptions: one that means a statement had the value False, one that means a statement had the value True, and one that signals the end of an iteration.

Various pieces of processing state are stored in global variables. Of these, the most important is the `PSmode` variable, which holds the indication of which processing mode is active at any point in time. Also significant is the `PSblock` variable, which holds an indication of which block type (Or or And) is active. This information is needed in order to determine what action should be taken in various cases.

Each procedure that implements a descriptive statement finishes by returning the result of invoking a common procedure called `value`. The `value` procedure is invoked with an indication of the value that has been determined for the statement. There are three possibilities: True, False, and Error. The third case is used when an error is encountered. The `value` procedure takes one of three actions based on the value indication it receives, and the current processing state:

1. If Error was indicated, a normal error exception is raised. Such an exception will propagate all the way to the top level.
2. In some cases, `value` simply returns, causing processing to continue with the next Prescription statement.
3. In other cases, `value` raises one of the special exceptions to alter the flow of control. The special exceptions are caught and handled further up the call tree.

In the context of an And block, statements must be processed sequentially until either all have been processed, or one has the value False. Thus `value` will simply return on True in an And context, and throw the False exception on False. In an Or block, processing continues sequentially until a statement has the value False, so the return action is different. Simple return occurs on False (to cause execution to proceed), and the True exception is raised on True.

The remainder of this section is devoted to brief summaries of how the individual blocks and statements work.

4.2.1 and

The `and` procedure is very simple. It saves the old `PSblock` setting, sets `PSblock` to And, and then invokes the interpreter on the block. Any exceptions are caught, so control is guaranteed to return to `and` on termination of block execution. The saved `PSblock` value is then restored, and `value` is called to end. When no exception is raised, all statements in the block must have the value True, so the statement has the value True. Exceptions are simply re-raised.

4.2.2 or

The `or` procedure is a bit more complicated then `and`, due to problems of repair. Basically, the block is processed first in verify mode, regardless of the operative mode. This is necessary so that procedures in the block do not attempt repair before the need for it has been established. If the block execution terminates without an exception, all statements must have the value False, so repair is required if the original mode was repair mode. The mode is reset, and if repair is required, the block is executed again in repair mode. This will cause each procedure to attempt repair. In the context of an Or block, a True value for a statement will cause an exception to be raised. That exception has the effect of terminating block execution.

4.2.3 prescription

The `prescription` procedure implements prescription definition. Both definition and activation will be described here. When a prescription is defined, a new Tcl procedure is created with the same name as the prescription. The argument list of the new procedure is just the parameter list of the prescription. The body of the new procedure consists of a call to a special pre-process procedure, a call to `and` with the prescription body, and a call to a special post-process procedure. When a prescription is defined as narrowed (with the `-narrow` switch) the call to `and` in the body is enclosed in a call to `narrow`. Since a new procedure is defined, activation of the prescription may be written simply as a normal Tcl command, as has been illustrated previously. The special procedures called in the body perform mostly administrative and data management operations. They do not normally alter the flow of control. Note that exceptions generated during execution of a prescription body propagate to the block that contained the prescription activation.

4.2.4 narrow

The `narrow` procedure is very simple. It merely saves the current mode indication, sets the mode to verify, and executes `and` with the block. Any exceptions are caught so that the mode can be reset, but exceptions are merely propagated upwards. Thus the `narrow` procedure does not directly affect the flow of control.

4.2.5 logical

The `logical` procedure is straightforward, although there are a couple of independent cases. If the attribute/operator pair are omitted (by passing empty strings), the expression part is simply passed to the Tcl expression evaluator. The result determines the value of the statement. In the prototype implementation, processing of non-repairable logical statements in repair mode generates an error. When all parts of the statement are supplied, the procedure still passes the expression to the Tcl evaluator, but then obtains the value of the referenced attribute, and compares that value to the value returned by the evaluator. If the result is False then a repair procedure is invoked to set the attribute in question to an appropriate value.

4.2.6 forall

The basic strategy for both `forall` and `disallow` statements involves iterating through the members of the specified collection. For some `require` statements, iteration is also necessary. A common internal procedure handles all three iteration situations. Iteration involves the same basic steps regardless of which type of statement is being processed. Objects are extracted from the collection one at a time. Each object is bound, in turn, to the given variable name, using a Tcl command. Each object is evaluated to determine whether it meets two constraints:

1. The type constraint given by the type specification for the variable.
2. The supplied constraint expression (applies only to `forall` and `disallow`)

For each object which meets the constraints, the block is executed. The actions taken, based on the results of executing the block, vary depending on the type of statement. In the case of `forall`, a False exception for the block implies a False value for the statement. Otherwise, iteration continues until there are no more objects, at which point the statement must have the value True. To handle statements over the closure of collections, each object is checked to determine whether it is a collection containing the same type of objects as the original. Such collection objects are added to a queue of collections from which objects are retrieved. Iteration ends only when the queue is empty.

The `forall` procedure does not implement any special repair actions, because repair must always be performed by procedures contained in the block.

4.2.7 disallow

The `disallow` procedure calls the iteration procedure whose basic operation is described above. For `disallow`, the only special action occurs when an object is found for which the body has the value True. In repair mode, an attempt is made to destroy the offending object, and continue with the iteration. In verify mode, iteration stops immediately because the statement is known to have the value False. For a `disallow` block, execution always proceeds in verify mode, to prevent statements contained in the block from attempting repair operations.

4.2.8 require

There are two cases for `require` to handle. When no identification value is provided, the statement is implicitly narrowed, and processed by iteration using the common iteration procedure described above. Iteration terminates for `require` as soon as an object is found for which the body has the value True. If no such object is found, the statement has the value False.

When an identification value is provided, an entirely different algorithm is used, one that does not involve iteration over the contents of the collection. Instead of iterating, the identification value is used to search the collection for a candidate object. If none is found and repair mode is active, repair is initiated with the creation of a new object having the required identity. When an object is available (either because it was found in the first place, or because it was created through a repair operation) it is bound to the variable, and the block is executed. The value of the block then determines the value of the statement.

4.3 Objects and Data

The base Tcl language uses strings as (almost) the only data type. The Prescription language requires support for more sophisticated data structures. It was necessary to have Tcl variables hold references to objects, and then to be able to deal with attribute references involving variables as was illustrated earlier. Sometimes variables also need to hold simple string or numeric values, which Tcl can intrinsically manage.

One way to deal with references and data is to use variables exclusively to hold object references. Objects could be created for each simple value, and the values handled through references to objects. While this approach has the benefit of consistency, it causes a lot of inconvenience when simple values are used.

An alternative approach is used in the prototype. Variables may hold both ordinary Tcl values *and* object references, and a special structure is used to distinguish object references. Object references begin and end with a special character sequence that is deemed to be uncommon in ordinary values[1].

Attribute values are used throughout prescriptions. Consider what happens with a reference such as the following.

```
$obj.one.two.three
```

The Tcl interpreter is capable of performing value substitution for the Tcl variable reference (`$obj` in this case). The interpreter leaves the remainder of the string alone. In order to use attribute values, it is necessary to invoke a procedure which can interpret the string left after substitution by the Tcl interpreter. The procedure is called `val`. What `val` does is take a string consisting of an object reference (distinguished by the special prefix and postfix string) followed by a period-separated list of attribute names. It dereferences the string by performing **`getAttribute`** operations, and returns the final value. For convenience, `val` will return an ordinary string unchanged. The typical use involves the in-line procedure invocation mechanism of Tcl, and is written as follows:

```
[val $obj.one.two.three]
```

1. The sequence presently consists of the ASCII vertical tab character, but could be easily changed.

Many procedures that are part of the implementation expect to receive object or attribute references as arguments, and deal with them appropriately. It is only necessary to use `val` explicitly in situations where an ordinary value must be supplied (e.g. in a Tcl expression).

The format of object references, storage of instance data, and invocation mechanisms are all described in the next section.

4.3.1 Object System Implementation

The base Tcl language provides no direct support for object-oriented programming. Although there are extensions which add object-oriented support, a small, custom object system is used in the prototype implementation. A custom solution has the benefit of supporting the features which are needed for this application.

A single inheritance class structure is supported, as described earlier. Per-class instance data is allocated transparently to the class implementations. Remote invocation is supported. These basic features are all common in object-oriented languages.

One unusual problem in this application is the problem of *object recognition*. The purpose of the software abstractions called objects is to represent components of managed systems, such as files, directories, or processes. The objects, however, are only created as needed. For instance, if a prescription describes a particular file (with a `require` statement), an object will be created to represent that file, although objects may not exist to represent other files in the same directory. During processing of a prescription, however, the same component may be described multiple times (e.g. by `require` and also by `forall`). The system must recognize any component for which an object has already been created, so that on subsequent reference the existing object is used (with the proper model state) and a new object is not created. By object recognition, the system avoids having multiple mutually inconsistent models for a single managed component.

In order to efficiently solve the recognition problem, objects are referenced by *global identifiers* (GIDs) which are derived from the identification information of modelled components. When a component is referenced, the software can simply generate the GID, then check to see whether an object already exists for that component. This approach to object references differs from regular practice in object-oriented languages.

In the prototype implementation, an object reference is composed of three parts:

1. Identifier of the class of which the object is an instance.
2. Identifier of the machine on which the object resides.
3. GID for the object.

These three items, along with the class definitions, provide enough information to invoke a method. It may seem obvious that the class identifier should somehow be stored with the instance data, and not passed around in references. That would introduce a small problem for the **claim** method, which modifies the reference to an object when claiming it. The machine identification could probably be handled differently as well. In the prototype, the three parts of a reference are combined in a Tcl list (which is just a string with a particular format). The generation of GIDs from components must be performed by the class implementations.

Instance data are stored in a single array on each machine. The array is a Tcl associative array[1], indexed by a combination of GID and class. This arrangement provides, for each object, one array slot for each class in the object inheritance hierarchy. The slot may contain any string, and so may be structured as a list to hold multiple items. A specific slot is not arranged for each declared attribute, as might be expected. This is because the attributes apply to the modelled component. There is no reason that the mapping between stored instance data and supported

1. The associative array of strings is the only data representation supported in Tcl, apart from strings themselves. Access to individual slots is implemented efficiently.

attributes must be 1:1. Instead, a class implementor is free to organize the instance data in whatever way seems most appropriate. For each object, one instance data slot is reserved for use by the prescription processing code (the slot associated with Object, the minimal root class). The reserved slot is used for the central records maintained to deal with deferred execution, as explained in Section 3.10.1.

Classes are defined by the use of procedures `class`, `method`, `class_method`, and `attribute`. In the prototype implementation, the explicit declaration of attributes is entirely at the discretion of the class implementor. If attributes are explicitly declared, then the processing code will use the declarations to check references, otherwise the class implementation must assume more verification responsibility. Since Tcl has all the facilities required of a class implementation language, method code can be included directly in the method definition, as illustrated by this example.

```
class FriendlyClass Object {} {
   attribute greeting String {}
   method greet {name} {
      return "Hello there $name"
   }
   class_method identify {} {
      return "This is the Friendly class!"
   }
}
```

The explicit declaration of the class of contained objects is another unusual feature of this object implementation. The prescription processing code uses the contained object information to process statements that relate to collections (`forall`, `require`, `disallow`).

The method invocation mechanism is a straightforward implementation of dynamic method lookup. When a method is invoked, hidden parameters `self` and `gid` are passed in, to provide instance information. The invocation code also arranges for local variable `idata` to be bound to the appropriate instance data array slot (using a Tcl feature). A procedure called `super` is provided to permit the implementation of a method to invoke the superclass implementation of the same method. The invocation mechanism for class methods is very simple, as there are no class objects. No inheritance-based search is performed to find a class method; it must be defined on the specified class or an error is raised.

4.3.2 Deferred Execution

The subject of deferred execution has already been addressed in considerable detail. The central record keeping code consists of a set of Tcl procedures with the same names as the methods in the standard set. These procedures are collectively known as the *Object layer* of the implementation.

4.4 Prototype Classes

In order to handle the desired examples, a number of classes were written as part of the prototype implementation. This section presents the interesting details of the prototype classes. Note that these class implementations are very specific to SunOS 4.

4.4.1 Files

Classes were written to model Unix files, directories, and symbolic links. The inheritance hierarchy for the classes involved is shown in Figure 1. The primary class is the File class, which

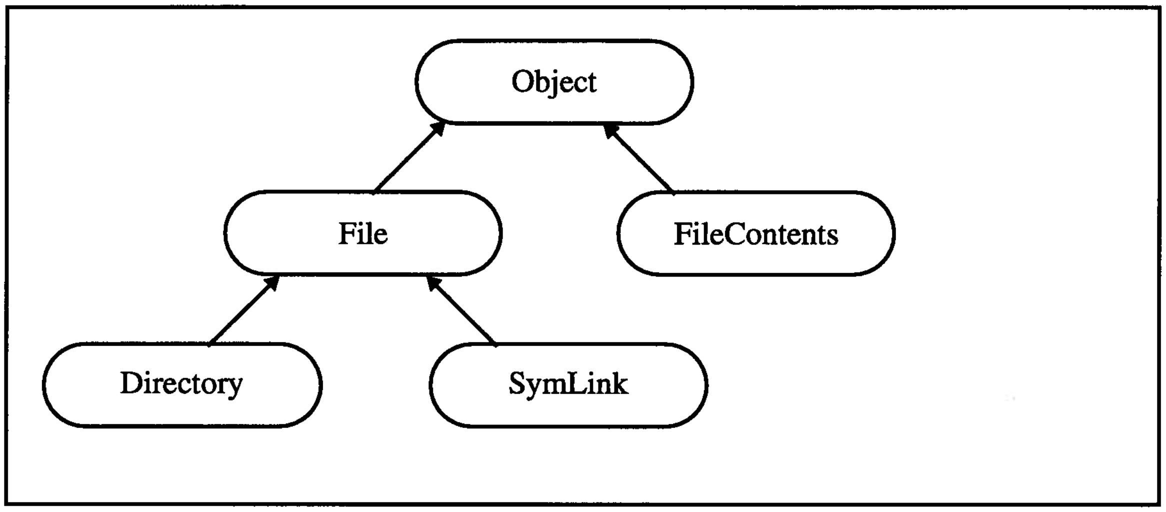

Figure 1 – File Class Hierarchy

models all Unix files. A number of attributes such as name, fullPath, size, and group are supported.

All file objects use the same GID format: /File/*hostname*/*pathname*. Since Unix files may have different names (due to links in the filesystem), this is not an adequate GID format for a production system. It is convenient for prototyping, since the file path can be extracted.

The only methods which are actually implemented in class File are **printRef**, **getAttribute**, **getDefault**, and **setAttribute**. For all other standard methods, either the default implementations in class Object are adequate (regular methods) or the method will never be invoked (class methods).

The FileContents class exists for the `contents` attribute of File objects. The primary method implemented in class FileContents is **isDeepEqual**. Through this method, the `contents` attribute may be used to describe one file as a copy of another.

The Directory class is quite substantial, since a Directory object is a collection of File objects. The **claim** class method and the various collection related methods are implemented in class Directory. One of the trickiest issues in the Directory implementation concerns the attribution of new objects extracted through the getNext method. Normally, such objects are created as instances of File, but directories are immediately declared to be of class Directory. This is necessary in order for the code that handles closures to work correctly. Key values for `require` statements (used in invocations of **find**) are simply file names, so lookup is quite efficient. In addition, the implementation of iteration supports optimization for a very particular form of constraint expression involving the `globEQ` function. This optimization is extremely primitive, to the point of being slightly incorrect, in a way that will be described in the section on limitations.

The SymLink class models symbolic links. The nature of symbolic links in Unix poses particular modelling difficulties. In most cases, a symbolic link object needs to be treated as the entity referenced by the link, not as the link itself. On the other hand, a symbolic link has an attribute which other files do not have, namely the item to which the link points. Due to this classification problem, the **claim** method is particularly important for SymLink objects. An object that models a symbolic link will only be declared to be of class SymLink if it is referenced in a statement which uses the SymLink type explicitly. The default value of the `ref` attribute of a SymLink object is always "/dev/null". In the prototype implementation, a SymLink is actually created pointing at the null device initially. In practice, creation should always be followed by a **setAttribute** invocation which sets the link to point somewhere in particular. Setting the `ref` attribute involves destroying the link. Thus some efficiency could be gained by tracking the state in memory and only adjusting the managed system when commit is invoked.

4.4.2 Fstab and Printcap Classes

Two Unix configuration databases, /etc/fstab and /etc/printcap are modelled by prototype classes. For each, there is a main class (e.g. FSTAB) that inherits from File, and a record class (e.g. FileSysRecord) that inherits from Object. The implementation of these four classes follows the strategy of directly tracking the component states. The constructors for the collection objects read the file data and store it in memory. Changes are made to the in-memory copy, and the entire file is rewritten on commit (which is only invoked if modifications were made).

Since the collection classes are subclasses of File (in recognition that the databases are stored in files), the GIDs follow the format for File GIDs. The GIDs for records are composed from the hostname of the file, the path of the file, and a key value for the record.

The key values are appropriate to the databases in question. For class FSTAB, an identifying value must be an ordered pair (*servername*, *filesystemname*). For the Printcap class, an identifying value must be the name of a printer entry. Neither class supports any optimizations for iteration.

4.4.3 Table Classes

Classes for specification tables are an important part of the implementation. Inheritance is used effectively with tables. There is a Table class and a TableEntry class. Both of these contain complete generic implementations. The class created for each specific table, and the class created for entries of each specific table, are empty subclasses of the two generic classes.

As noted earlier, the prototype implementation of tables relies on text files with a simple structure. The data for the table is read from the file specified by the *filename* parameter to the `table` definition statement. The expected format is a sequence of records delimited by newlines, each containing the same number of fields delimited by the *field-delimiter* string.

Fields in a table all have declared types. When a value is requested for a field of a fundamental type (using **getAttribute**) the value is returned exactly as it is written in the file. Alternatively, a field type may be the name of another table or class, or a list containing values of some other specified (non-list) type. If the field type involves a class, the **getAttribute** code treats the value recorded in the file as a foreign key identifying an object in the master collection for the specified class. The master collection object is retrieved by invoking the class method getCollection. When the field value is requested, the referenced object is obtained and returned. This mechanism is used primarily in the case of fields which contain foreign keys for records in other tables. When the special @ syntax is used, reference translation is suppressed and the literal value is returned.

The table implementation does not support any form of modification to tables through repair processing, because the tables are considered part of the specification, not part of the managed system. Prescriptions may be written to discover errors in tables, but automated repair will not work. It is perfectly reasonable to develop classes for tables as a components of a managed system, as the Printcap and Fstab classes demonstrate.

4.4.4 Machine Class

Machines are also modelled by a class. A table is required to define the known machines to the system. The attributes of a machine are the attributes of the entry for the machine in the definition table, plus attributes whose values are particular objects on the machine. For instance, the `root` attribute is the object which models the root directory of the machine. The primary purpose of the Machine class is to provide access to objects on a machine. The handling of machines is explained in more detail in the next section.

4.5 Distribution Facilities

The configuration management problem addressed by this thesis is fundamentally a distributed problem. The prototype implementation has a basic set of mechanisms for dealing with distributed systems. As language validation was the primary objective, the distribution scheme is relatively weak.

One central machine must be designated as the *master*. It is this machine which must have accessible copies of all the Tcl code, as well as direct access to the database files and configuration descriptions. All other machines (called *slaves*) require only TCP/IP reachability[1] from the master, and an appropriate Tcl interpreter. No shared filesystems are assumed between machines, and if available they are unused.

Interaction between machines is supported by the DP extension [2] to the Tcl interpreter. This extension provides a powerful RPC mechanism by which Tcl commands may be executed on remote machines and the results returned. Both master and slave machines act as RPC servers. On slave machines, starting the server is the only local initialization step that must be performed. All necessary code is downloaded into the interpreter automatically by the master when the slave is first accessed. Thus the part of the configuration management system that runs on every machine as small and generic.

In order to support distribution, a database table must be provided to define machines. The table may include any attributes desired, but must have name and aliases attributes. A procedure

1. Reachability is assumed to be reflexive.

called `machine` is invoked to create Machine objects based on the contents of the table. Initially the objects that model machines are all declared to be of class MachineProxy (a subclass of Machine).

Access to machines represented by MachineProxy objects is avoided by means of a hack. Whenever a prescription is activated with a parameter that is a MachineProxy object, the activation is elided, and the prescription is summarily declared to have the value True. This simple technique permits processing to proceed even when some known machines are inaccessible. The technique seems to work remarkably well in practice, but would distort the logical evaluation of some prescriptions unacceptably.The procedures `engage` and `disengage` are provided to enable and disable access to machines respectively.

A Machine object is always created on the machine which it represents. The master simply keeps references to the Machine objects on hosts around the network. A special class with only one instance (AllKnownMachines) is provided as a collection.

The processing of prescriptions is fundamentally centralized in the prototype implementation. In fact, all statement processing actually occurs on the master machine, with the slaves accessed only to perform operations on objects which are not on the master machine. Objects which model managed components (as opposed to representing database contents) always reside on the machine which contains the modelled component.

The prototype implementation follows a strict push model for managing a network of machines. This helps achieve synchronism, since whenever a change is made to the master configuration database, all machines can be immediately updated. When a pull model is used, it is more difficult to arrange for machines to be updated in a timely fashion after a configuration change. On the other hand, there are situations in which a pull model is more appropriate, as in the case of a machine returning to normal operation after having been turned off for an extended period. Using the prototype implementation, a machine could send a request for engagement to the master, then cause the master to process prescriptions again, but that would affect all active machines and involve a lot of redundant work.

The push approach described here could easily be adapted to eliminate redundancy. The master would need to keep track of the state of various machines, with respect to prescriptions that have been activated. When a slave contacts the master after having been down for awhile, the master could respond by performing the prescription processing required for that slave.

In the context of a distributed system, a number of standard problems arise. For instance, there is the problem of a single point of failure at the master. There is also the problem of network partitioning. These issues have not been addressed in this work.

Ideally, a production system would be much more distributed in nature than the prototype implementation. The central base of prescriptions and database descriptions should be automatically factored to produce smaller descriptions of the configuration of each individual machine. The prescriptions for each slave could then be transferred to it and processed locally as required. A central processing capability could easily be preserved for those machines which are not suited to local processing for some reason (e.g. X-terminals). Updates for most machines in this case would consist of prescription updates, and the master machine would primarily play the role of a specialized database server. In such a scenario, it is conceivable that the complete description of a machine configuration would be a combination of descriptions of different types, with some parts specified locally, others controlled centrally.

4.6 Administrative Support

A couple of administrative mechanisms support the rest of the implementation. The logging mechanism has already been mentioned. A simple trace mechanism is also included; it permits helpful messages to be produced when something goes wrong.

During the first processing phase a log is maintained of the repair operations that are to be performed in the second phase. The log is stored as a Tcl list. Each entry consists of a type field, an identification of the code that generated the entry, a reference to the object which the entry is about, and data describing the operation. The start and end of processing of each prescription is recorded in the log in order to identify the context of recorded operations when the log is reviewed by a human. The prototype implementation contains a minimal facility for printing the log, along with code to perform the operations recorded in the log. One shortcoming of the logging mechanism is that a log entry about an object modification is recorded on the machine on which the object resides. There is no adequate central mechanism for accessing and manipulating these logs which are spread across various machines.

The tracing facility is provided in order to gather enough information to create informative messages for an administrator when something goes wrong. When an error occurs (or a prescription is determined to have the value False) it is useful to know what statement was being processed. The tracing mechanism is called in various places to accumulate a prescription activation trace (the equivalent of a procedure call stack trace). The stack trace maintained by the Tcl interpreter is not useful, because it contains too many frames of implementation procedures. The prototype trace facility is very limited, but is helpful nonetheless.

4.7 Issues, Limitations, and Problems

This section is devoted to descriptions of a range of discoveries from the implementation exercise. Some of the issues and limitations have already been mentioned, but are explained in more detail here. Many of the topics in this section are general, and not solely applicable to the prototype implementation. Note that performance issues are not mentioned here, since they are covered in a separate section which follows.

4.7.1 Inter-class Dependencies

In the prototype set of classes, there are frequently strong dependencies between the implementations of different classes. Some of this is natural and seems unavoidable. For instance, all classes that model files share a common GID format. Other dependencies may reveal architectural problems. For instance, the **create** method of the Directory class performs creation operations for instances of the classes File and SymLink as well as Directory. The general problem may be the fact that no class methods are invoked to accomplish creation. Methods are only invoked on the collection object which will contain the new object.

4.7.2 Multiple Architecture Support

The prototype implementation and trial example do not adequately explore the problems of supporting multiple architectures. There are two ways in which the language and model should be well suited to multiple architecture support. In the first place, prescriptions can be easily conditionalized based on architecture, using the `if` statement. Furthermore, there is a good possibility that minor distinctions between similar systems can be hidden behind the object abstraction. For instance, two variants of Unix might have slightly different forms for the printer definition database, but the differences might be at a syntactic level. If so, they can be handled entirely by different class implementations. I expect that separate class implementations will be required for each system, even though there may be a common abstraction that all the implementations support.

4.7.3 GID Problems

The object recognition problem creates the need to use global object identifiers that are unique, and that can be generated correctly by looking only at a managed component. The GID form used in some of the prototype classes is inadequate for a production environment. Recall that the File class (and descendents) use GIDs based on file name. The problem with that choice has already been mentioned. A more appropriate GID structure for Unix would be based on the (host, device, inode) triple. The problem with such a GID form is that mapping from the identifying triple, to a name that can be used in operations on the specific item, is non-trivial. The solution is to maintain the path information in the object instance data, but that introduces the problem of multiple paths to the same item. The problems related to links and GIDs are discussed in the next section.

4.7.4 Component Classification

The basic problem of component classification arises with symbolic links, as has already been described, and also arises with hard links and possibly other structures. The managed system has these subtle features which create possible ambiguities. How are we to model such things? The prototype solution is not entirely satisfactory. With a better GID scheme, as discussed above, a better scheme for handling files referenced by symbolic links could be introduced. When referenced as a file or directory, a symbolic link should probably be bypassed, and an object generated to model the referenced item. Then there is a problem with classifying an object as to the collection in which it resides. If an object needs to be recognized as being contained in every directory from which it can be accessed, every link must be found; such a search is a non-trivial exercise.

4.7.5 Implicit Side-effects and Dependencies

Implicit side effects cause problems. Inter-attribute dependencies cannot be handled using the standard method set, as explained in Section 3.10.1. Another form of implicit side effect occurs with the description of entries existing in an fstab database. The purpose of such an entry is to define a filesystem, and often all defined filesystems are mounted when possible. We can assume, therefore, that modifications to an fstab file are intended to imply changes to the set of mounted filesystems. If the implication is correct, then the implementation of the **commit** method in the appropriate class should perform mount/unmount operations. Here we encounter another problem with deferred execution. The problem is that the implicit side effect of creating an entry in the table may be depended upon by prescription statements processed after the statement that describes the entry. The Object layer attempts to provide a generic solution to the deferred execution problem, but it cannot generically handle implicit side-effects like the mounting of a filesystem. With this particular case, we can observe that mounting a filesystem is an operation that may be easy to reverse. Thus the implementation could be changed so that the prepare method on the FSTAB object actually goes ahead and performs the mount operation. That fixes the problem of the dependency on an implicit side-effect, but complicates the problem of maintaining atomicity. Where the present implementation has only a *do*-log, the new implementation would require support for an *undo*-log as well. Then the mount operation could be recorded in the log, to be undone should that be necessary to maintain atomicity.

4.7.6 Atomicity, Distinct Machines, and Parallelism

Atomicity and related matters are some of the most problematic issues surrounding the Prescription language. Atomicity is clearly an important principle, yet it has awkward ramifications as it has been defined up to this point. The problem is that, in typical situations, errors or inconsistencies tend to be propagated upwards to the top level. Consider the situation of a set of prescriptions organized in an activation hierarchy as in Figure 2. If only And blocks are involved, then a

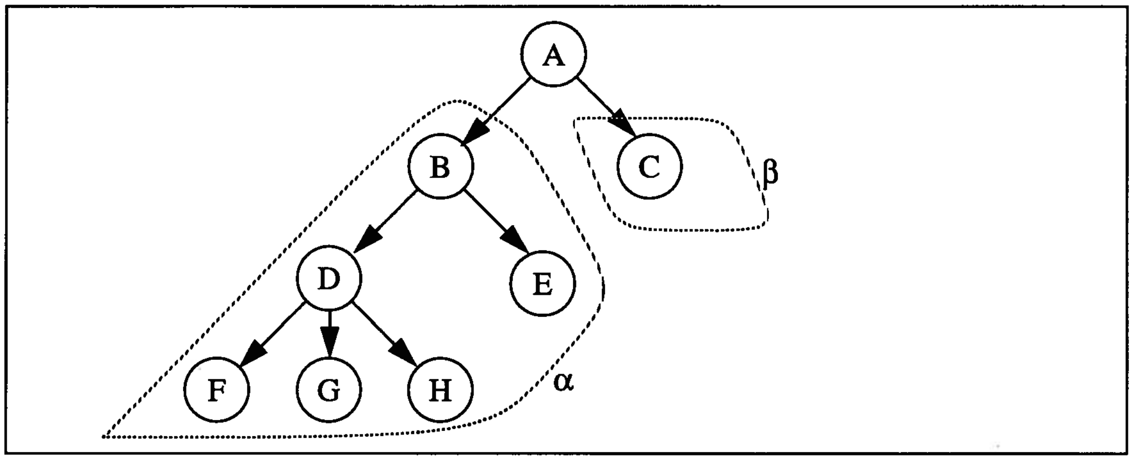

Figure 2 – Activation Hierarchy

statement determined to have the value False in prescription G will cause G to be declared False, which will cause D to be declared False, which will cause B to be declared False, which will finally cause A to be declared False. Perhaps, however, the specification given by D is partly independent of the specification E, and B serves a grouping purpose. In that case, processing of B should not stop when the problem with G is detected. Also the failure of G should not necessarily cause the repair actions required for F to be skipped (or undone).

Further problems arise when we introduce machine boundaries, since machines are independent in important ways. Suppose that the bubble labelled α, in Figure 2, surrounds prescriptions which describe the configuration of one machine, while bubble β surrounds prescriptions which describe a different machine. In this case, we can ask whether a problem with machine α should prevent processing on machine β. The problem of dealing with machines that may not be reachable has already been mentioned. The solution used in the prototype implementation is unsatisfactory.

The language lacks features for properly describing the places where atomicity is required, where machine boundaries must be ignored, and where parallelism is acceptable. Prescriptions do the job that they were intended to do, but they do not provide enough structure for a complete configuration management system. If a prescription is analogous to a procedure, then what is missing is something analogous to a program or a package.

4.7.7 Exclusivity and Change Management

Another tricky issue is the handling of parts of configurations that should be described *exclusively.* The Prescription language is primarily oriented towards *inclusive* descriptions. If a file is described as existing in a particular directory, there is no implication that the described file is the only file in that directory. In many situations, these are the right semantics. In other situations, the implication may be desirable. For example, consider the case of a prescription that describes printer definitions based on entries in a database table. The general skeleton of the description would look like the following:

```
forall P PrinterDesc $table {} {
   require N PrinterRecord [val $P.name] $m.printcap {
      ...
   }
}
```

The `require` statement describes the inclusion of a printer definition in the printcap table, and the `forall` statement therefore describes the inclusion of a definition for each entry in the table. It does not describe the exclusion of all other entries which might exist, although that may be desirable. It is possible to write a `disallow` statement that does describe the exclusion of all printer definitions that are not based on a table entry. The problem is that such a description is awkward to write and slow to process. Worse is the situation in which different printer definitions are described by different prescriptions. It may become very difficult to write a `disal-`

low statement that is correct. It would be much easier if a particular collection could be marked as exclusively described.

The problem of exclusivity arises with specification change. Suppose, for instance, that a central database describes various printers that must be defined on machines, and prescriptions describe the machine configurations based on the information in the database. For each printer, a printcap entry must exist along with a spool directory. Now suppose that a printer is retired and not replaced, so that the name and definition are no longer valid. It should be the case that the definitions for the old printer, in the printcap files of various machines, disappear. It should also probably be the case that the spool directories for the old printer are removed. One way to manage this would be to create `disallow` statements at the time when the printer is removed. While adding statements would achieve the desired effect, this strategy would cause descriptions to grow, over the long term, in a practically unbounded fashion. The `disallow` statements would have to remain part of the configuration description until administrators could be sure that they had been processed on every machine.

The situation just described seems to indicate a need for a simple procedural administration mechanism. What would be most helpful, however, is a system capable of automatically determining the complete sequence of operations that should be performed to get from one configuration state to another. Then an administrator could simply make a central configuration change, without worrying about describing garbage that should be removed, or specifying commands to remove the garbage.

The problem of automatically managing change is a significant one in a variety of software contexts. In configuration management it seems to be a major roadblock on the path towards automation.

4.7.8 Optimizing Object Selection

The `forall` and `disallow` statement types include selection expressions for identifying the objects from a collection to which a statement applies. Ideally, these expressions should be able to be used by class implementations to limit the search scope in structured collections such as the closure of a directory. This is done in a limited way in the prototype implementation. Two problems arose and are explained here.

The first problem is that the class implementation must be able to extract suitable information from the expression. Remember that the expression is a normal boolean expression involving the variable declared in the statement. Extracting the right information is not an easy task in general. In the prototype, an optimization is applied only when an expression has a particular, restricted form, from which the right information can be easily removed. If an implementation used traditional parsing techniques, it might be easier to deal with general expressions. Alternatively, it might be better to leave the form of the expression entirely up to the class implementor. While that approach would simplify optimizations, it would also create a lot of inconsistency.

The second problem in the prototype is that the simple form of optimization that is used (globbing through a directory structure), results in objects being "found" in the wrong collection. A search for a file a/b/c in directory /top is certainly fast, but if the file is found, the processing code erroneously treats the file as being contained in the /top directory. This is a problem for the `disallow` statement, since the file object might be slated for removal, and the state of the wrong collection object would then be modified.

When closure processing is performed outside the class implementations, this same problem with selection optimizations arises. An optimization may permit fast isolation of candidate objects, but they may not all be contained in the collection under examination. The class interface for iteration needs to be adjusted to properly handle this situation. It should provide a way for the class implementation to pass back a collection object with each selected object. The main processing code then needs to avoid closure traversal, and rely on the class to produce the correct objects.

4.7.9 Under-specified Creation

There are various kinds of objects for which creation with default values for most attributes does not make a whole lot of sense. The prime example of this is the symbolic link. When creation is determined to be necessary (in processing a `require` statement), only the identification of the symbolic link is known. To create a link, however, the referent must be specified. The prototype implementation creates a link pointing to /dev/null, but it is probably better to defer creation until more information can be accumulated. The **commit** method is useful for this purpose.

4.7.10 Attribute Changes that Change Identification

Some attributes have significance in object identification. If modification to such attributes is permitted, awkward problems may arise. For instance, a modification called for during processing of the block in a `require` statement might cause the object in question to fail to match the identification value in the statement.

It is common to find that the GID of an object depends upon the value of certain attributes. If those attributes can be modified during repair, the possibility of a GID change for an object must be considered. Some attempt was made to prepare for this in the prototype implementation, but the ramifications were never fully explored. It may not even be necessary for the GID to change, as long as there is no possibility that the object in question will be encountered under the new GID. This may be the situation when the object in question is an entry in a file, and cannot be accessed except through the file. If the GID does not change, however, there is a possibility of GID collision when new objects are created.

4.7.11 Reference Values in Tables

As has already been described, central database tables are expected to contain fields which reference records in other tables. The **getAttribute** method for the TableEntry class automatically translates such references to the referent object, unless the special @ syntax is employed. The find method on the Table class, however, does not correctly deal with object-valued key fields, so the key value must be used. Lookup by value stored is actually more efficient than lookup by referenced object would be, but the inconsistency is disconcerting.

4.7.12 Static Analysis

Since the Tcl implementation is interpreted, there is no real opportunity to perform static analysis on specifications. This is a significant limitation, as there are a number of potential uses for static analysis:

1. Static type checking. Type checking for prescriptions would have all the benefits of type checking in ordinary programming.
2. Static identification of repairability problems.
3. Static factoring of prescriptions to permit distributed and parallel processing.

4.7.13 Aliases

It is common, in practical distributed systems, to use multiple names for the same entities, particularly for machines. Use of functionally meaningful names limits the scope of change when a function is transferred from one machine to another. Extensive use of aliases, however, poses some problems for a prescription-based management system.

In the first place, it is important that object recognition is preserved. In the prototype, machines and printer entries may be looked up by any alias, but the same object is always returned for names which map to the same entity. When an aliases attribute is modified, the problem of changing identification, discussed in Section 4.7.10, arises.

Secondly, it is sometimes awkward to organize prescriptions so that the configurations of machines use the correct aliases, rather than the canonical names. For instance, if a filesystem is declared to be exported by a machine "mainserver", the correct Machine object must be referenced. The definition of the filesystem on any client, however, should use the alias "mainserver" regardless of the canonical name of that machine. The @ syntax described in Section 3.3.2 is useful for solving this problem.

It is not always clear how aliases ought to be handled. Entries in a printcap file, for instance, include several equivalent names. In the prototype implementation, the first one is taken as canonical, and the others as aliases. A `require` statement may identify an entry by any of the names, but if an entry is to be created, the supplied identification value will become the *first* name.

4.7.14 System Inconsistency

There is no guarantee that a managed system is in an appropriate state when prescriptions are processed. It may, in fact, be seriously damaged. The prototype implementation is not at all robust when dealing with problems such as file format errors. It is not clear how such problems should be handled, but it would be nice if a configuration management system could cope with them.

4.8 Performance

The implementation described in this chapter was produced as a proof-of-concept demonstration, so performance was not an issue in the design or development. The system is extremely slow, in fact. There are good reasons why this is so, and why it should not be a significant concern.

First, the standard Tcl interpreter is not very efficient. All code, and almost all data, is maintained internally in strings. No compilation techniques are used. In a typical program, the interpreter parses and evaluates many code segments many times each. The Tcl language is a challenge to handle efficiently because of its highly dynamic nature and the extensive use of calls to C code in the implementation. Nonetheless, there are things that can be done to provide impressive speed improvements. In tests presented for one compiler effort, speedup factors between 1.30 (pessimistic) and 12.17 were reported [3]. There is reason to believe that even more impressive results could be obtained with further work.

The second cause of poor performance is the nature of the prototype implementation. It is written entirely in Tcl, which means that performance is highly dependent on the interpreter. The data management operations involve many string manipulations, such as regular expression tests to check whether a string contains an object reference. Finally, the implementation was written with much more attention to clean design than to efficient execution. Little optimization has been done. One way to substantially improve the implementation is to rewrite many pieces in C. The method invocation mechanism would be a good initial candidate.

The poor performance that has been observed should not be a major cause of concern for a few reasons. First, the implementation suffers from the obvious problems noted above, for which there are promising solutions. Also, the Tcl code could be abandoned in favour of a carefully constructed implementation in some compiled language. The most important observation is that the processing of prescriptions is not inherently inefficient, as can be seen by reviewing the algorithm descriptions in Chapter 3.

The main potential problem for a production implementation is the management of relatively large amounts of data. Given the capabilities of today's machines, and the excellent prospects for distributing the work, even this problem is unlikely to be serious.

CHAPTER 5 Experiment

Any new approach to configuration management, such as the one presented in this thesis, needs to be validated by application to real-world situations. This chapter is a report on an experimental application of the new model to the configuration of client workstations in the department of Computer Science at the University of British Columbia.

The experiment tested both parts of the model presented in Chapter 2:

1. The viability of the specification part was tested through the attempt to apply it to a large, real-world environment.
2. The validity of the automation part was tested by running the prototype implementation on a significant, real-world specification.

These two parts of the experiment are described separately. The chapter concludes with a unified analysis of the results.

5.1 The Specification

To evaluate the model and the Prescription language, a significant, practical problem was needed. Important aspects of workstation configuration in the Computer Science department are described in a centralized database that is used by a locally developed software tool. This existing description provided an ideal test case. For the experiment, the same configuration information was expressed using tables and prescriptions. Not every detail was precisely duplicated, rather an alternate specification was produced with the same effect as the original.

The first step in the translation was a careful analysis of the configuration information, to isolate abstractions and identify the appropriate units of description. Based on this analysis, a set of tables was synthesized.

An example of an abstraction discovered is the *logical filesystem*. A large part of the centralized configuration description is devoted to filesystems that are imported via NFS[1] on client machines. In a number of cases, filesystems are organized into groups. All the members of a group are imported together. One member may contain a root directory for the group, containing symbolic links that reference parts of the other members through standardized mount points. With the Prescription model, this abstraction may be directly supported, through a table that defines logical filesystems in terms of individual filesystems and other data.

The second step was filling the tables with the information that describes the details of machine configurations. In the process, various limitations of the schema were discovered and eliminated.

Finally, a number of prescriptions were written, describing the configuration of the managed system in terms of the data in the tables. The prescriptions provide the interpretation of the

1. Network File System

tables, and specify details which are common across entries. The most interesting features of the final specification are summarized in the sections which follow.

The description consists of 14 tables (excluding the table which defines machines) and 16 prescriptions. For more detail than can be included here, refer to Appendix A.

5.1.1 Printers

The description of printers demonstrates relationship between tables and prescriptions. The Printer table holds entries which describe the various instances of printers. The table is defined by the following fragment (which assigns the table object returned by procedure `table` to the variable `Printers`):

```
set Printers [table Printer printer.table | { \
   {name String} \
   {server Machine} \
   {aliases List String} \
   {type String} \
   {note String} \
   {maxSize Integer} \
} {name} {} ]
```

Here are a few sample entries from the table:

Table 5: Printers

name	server	aliases	type	note	maxSize
hp306	hp306	{HP 4si (in CC306)} *c-hp4s-i,CC306	HP 4si	Room CC306	
lw106	lw106	garibaldi laserwriter {NEC Silentwriter (in CC106) 1} *c-nec,CC106	NEC Silent-Writer	Room 106	
cicsrlw	lwcicsr	lp lwc default {Silentwriter (in CC289)} *c-nec,CC289			10000

Table 5: Printers

name	server	aliases	type	note	maxSize
lw238	lw238	clinker clink *c-nec,CC238	NEC Silent-Writer	Draft Room 312	

The following prescription describes the configuration of a client machine which will have access to a printer defined in the table above.

```
#-----------------------------------------------
# PrinterDef - describes a remote printer definition that
# must be present on a machine
#
# Parameters:
#  m : Machine
#  pd : Printer (from table)
#

prescription PrinterDef {m pd} {
   global All
   narrow { or {
      require M Machine [val $pd.server.name] $All {}
      require M MachineProxy [val $pd.server.name] $All {}
      }
   }

   if { [strEQ $pd.server.name $m.name] } {
      # Printer should be locally defined on its server
   } else {
      # A PrinterEntry must exist
      require P PrinterRecord [val $pd.name] $m.printcap {
         logical $P.name s= $pd.name
         logical $P.aliases s= $pd.aliases
         logical $P.lp s= ""
         logical $P.rm s= $pd.@server
         logical $P.rp s= $pd.name
         if { [strEQ $pd.maxSize ""] } {
           logical $P.mx == 0
         } else {
           logical $P.mx == $pd.maxSize
         }
         logical $P.sd s= /usr/spool/print/[val $pd.name]
         logical $P.ty s= $pd.type
         logical $P.note s= $pd.note

         # The specified spool directory must exist
```

```
               global root daemon
               dir $m $P.sd $daemon $daemon 02755
            }
         }
      }
```

The prescription describes two things. First, it describes the printcap entry, which defines the printer to the client machine. Second, it describes a spool directory which must exist. Note that the prescription specifies values for some fields of the printcap entry that do not change from printer to printer (in this organization). Also, some values are derived from values given in the table. For instance, the spool directory path is derived from a standard root and the name of the printer. Administrators do not have to enter a spool directory for each printer they define, and they have assurance of consistent naming. Another important feature of the prescription is the fact that it can be safely applied to a machine which happens to be the server for the printer.

The `PrinterDef` prescription relies on a prescription called `dir`, defined as follows:

```
# --------------------------------------------------
# dir - describes a directory that must exist
#
# Parameters:
#  m : Machine on which dir must exist
#  path : String pathname of directory
#  owner : Integer owner uid
#  group : Integer group gid
#  mode : Integer permission mode
prescription dir {m path owner group mode} {
   require D Directory $path -closure $m.root {
      logical $D.owner == $owner
      logical $D.perms == $mode
      logical $D.group == $group
   }
}
```

A separate prescription is used because directory existence is specified in a number of contexts, and the prescription can be easily re-used. This is a simple example of the value of modularity.

5.1.2 Filesystems

The descriptions of filesystems and related items are considerably more complicated than the descriptions of printers.

A table describes individual filesystems that may be imported via NFS. The table records, for each filesystem, a name, a server, the path of the filesystem on the server, indications of whether soft mounting and quotas are required, a canonical name under which the filesystem should be accessible on the client (optional), a mountPoint (optional), and an indication of whether the namespace of the server should be made canonical.

A few prescriptions are involved in the description of importing a single filesystem. Most of the configuration does not apply on the machine which is the server for the filesystem, but namespace canonicalization does. To canonicalize a namespace, symbolic links must be used. If a canonical name is given for a filesystem, a link with that name is specified to point to the mount point of the filesystem. Supplementary tables contain entries that describe arbitrary links to be made into the filesystem, and sets of links to particular directories based upon the architecture of the importing machine. The fstab entry that is required for the filesystem is described, as is the existence of the mount point directory. If the mount point is unspecified in the table, then a prescription defines one according to a simple formula, in the same way that a printer spool directory is defined. The mount options are partially determined by the values given for the soft mount and quota fields in the table. The original configuration description includes alternate server aliases to be used by clients on particular subnets. These details are accommodated in a supplementary table that contains the name to be used for various server/subnet pairs.

As described earlier, the abstraction of logical filesystems is supported by a table that describes filesystem groups. Each group has a name, a list of filesystems that are part of the group, an (optional) indication of the filesystem that holds the root directory, an (optional) path to that directory within the filesystem containing it, and an (optional) canonical name by which the root directory should be accessible.

The prescriptions describe the import of a logical filesystem in terms of the import of each of the parts, plus the existence of an appropriate symbolic link for the root directory, if there is a root directory.

5.1.3 Services, Groups, and Assignments

To complete the specification, there are tables defining groups of machines, groups of services, and the assignment of services to machines and groups of machines. Both groups of machines and groups of services can contain other groups of the same type, so there is a lot of organizational flexibility.

A service group is composed of other service groups, printers, individual filesystems, and logical filesystems. In addition, a service group may be limited to a particular architecture. There are also supplementary tables which describe directories, symbolic links, and file copies that must exist when a service group is assigned to a machine.

Here is the top-level prescription describing the entire configuration, based on the tables and the other prescriptions which are defined:

```
#-------------------------------------------------
# main - describes state of entire distributed system
#
# The use of the name 'main' is arbitrary.
# The appropriate state is described in various tables.
# This prescription consists of assertions about the
# system in relation to the data in various tables
prescription main {} {
   global ServiceAssign
```

```
        forall S ServiceAssignment $ServiceAssign {} {
           # Each service is assigned to the specified group of machines
           AssignToGroup $S.group $S.service

           # Each service is assigned to the specified individual machines
           forall M Machine $S.individual {} {
              AssignServices [Machine $M] $S.service
           }
        }

         global MachineAssign
         # Each machine has the specified individual services assigned
         forall M MachineAssignment $MachineAssign {} {
           forall S ServiceGroup $M.services {} {
              AssignServices [Machine $M.machine] $S
           }
           forall L FilesystemGroup $M.logicals {} {
              ImportGroup [Machine $M.machine] $L
           }
           forall F Filesystem $M.filesystems {} {
              GuardedImport [Machine $M.machine] $F
           }
        }
     }
```

The configuration of most machines is derived from the groups in which the machines are placed. The prescriptions for printer and filesystem definition have special cases for servers. The reason for these is that a server may be part of any group, so it may end up being assigned a service which it provides. A server should not be configured to import a service from itself. The prescriptions handle servers automatically, so special measures do not need to be taken to exclude them from certain groups or assignments.

5.2 The Processing

In the second part of the experiment, pieces of the specification described above were tested using production workstations and the prototype implementation.

Initially, trials were conducted with small, distinct bits of specification. For example, the `dir` prescription presented earlier was activated with various combinations of values for the parameters. Consider the following example, in which the prescription is used to verify the existence of the directory /usr/bin, with standard permissions and ownership:

```
mytcl>activate { dir $san /usr/bin $root $staff 02755 }
True
```

In this example, the variable `san` contains an object representing a machine. The variables `root` and `staff` hold the ids of the root user and staff group. The string "`mytcl>`" is the interactive prompt of the interpreter. The word `True` is the response of the implementation indicating the truth value that was determined. With a slight change, the statement becomes False:

```
mytcl>activate { dir $san /usr/bin/ $root $daemon 02755 }
False
      Logical false for <Directory>san.cs.ubc.ca:/usr/bin>.group
 value=<2000>
 op=<==>
 compareVal=<1>

Prescription Activation Stack:

0: Prescription "activate" : <Top level>
1: Prescription "dir" : {Machine <san.cs.ubc.ca>} /usr/bin/ 0 1 02755
Statement and {
   require D Directory $path -closure $m.root {
      logical $D.owner == $owner
      logical $D.perms == $mode
      logical $D.group == $group
   }
}
Statement and {
   logical $D.owner == $owner
   logical $D.perms == $mode
   logical $D.group == $group
 }
```

In this case, the implementation produces additional information to help pinpoint the discrepancy between the specification and the managed system.

Gradually, more and more extensive specifications were processed. The trials revealed various problems and implementation bugs which were addressed along the way. Finally, two major trials were conducted, as described in the following two sections.

5.2.1 Setup Trial

The code for the Machine class was modified slightly to treat a test directory (/tmp/R/) as the root directory of a machine. The experimental directory tree was populated with just a few basic files and directories, on one slave machine. For instance, an /etc/fstab file was created, containing only comments and the definitions of local filesystems. The prescription `main` was then activated for repair mode processing on a master machine distinct from the test slave. The test was designed to simulate a situation in which a new machine is configured. The resulting log contained 999 individual repair actions. Here are the first few records from the log:

```
Create new object <Directory>ice.cs.ubc.ca:/tmp/R//nfs>
Create new object <Directory>ice.cs.ubc.ca:/tmp/R//nfs/faculty1>
Set <Directory>ice.cs.ubc.ca:/tmp/R//nfs/faculty1> attribute <group> to
<0>
Create new object <FileSysRecord <faculty1,/faculty1> in /tmp/R/etc/
fstab>
Set <FileSysRecord <faculty1,/faculty1> in /tmp/R/etc/fstab> attribute
<dir> to </nfs/faculty1>
Set <FileSysRecord <faculty1,/faculty1> in /tmp/R/etc/fstab> attribute
<type> to <nfs>
Set <FileSysRecord <faculty1,/faculty1> in /tmp/R/etc/fstab> attribute
<options> to <rw>
Set <FileSysRecord <faculty1,/faculty1> in /tmp/R/etc/fstab> attribute
<options> to <rw intr>
Set <FileSysRecord <faculty1,/faculty1> in /tmp/R/etc/fstab> attribute
<options> to <rw intr bg>
Create new object <SymLink>ice.cs.ubc.ca:/tmp/R//faculty1>
Set <SymLink>ice.cs.ubc.ca:/tmp/R//faculty1> attribute <ref> to </nfs/
faculty1>
```

These particular records cover the operations required for the machine to import the faculty1 filesystem, which is a part of the faculty filesystem group. Note that five of the operations involve the fstab entry that is required for the filesystem. The references to paths beginning with

"/nfs" appear because the table data and prescriptions were not updated to use /tmp/R/ as the root directory.

Finally, the recorded operations were executed to carry out the repair. The implementation skips operations which require privilege, but the necessary directories, links, and file copies were created. Also the fstab and printcap files were updated with the required definitions. Comments that existed in the files prior to repair were not preserved, because the implementation does not support comments.

5.2.2 Verification Trial

The prescription `main` was activated for repair mode processing on a master machine with two engaged slave machines. Due to the fact that there were differences between the experimental specification and the configuration of the machines, a number of repair actions were logged for both machines. The logs clearly identified the ways in which the actual configurations deviated from the specification. Execution of the operations specified in the logs would have repaired the machines to match the experimental specification. Many of the deviations were due to the fact that the experimental specification uses a slightly different convention for naming mount points then is used in the original configuration description.

5.3 Evaluation

The experiment was successful. A specification was produced for a substantial, real-world system. The prototype implementation processed the specification in multiple ways. The experiment was certainly not exhaustive, but did test the essential features of the model that is presented in this thesis.

The evaluation is not complete with the conclusion that the experiment was a success. A number of more interesting observations can be made. To begin, we can compare the new approach to the existing technology. The configuration description that was used as a basis for the experiment is input to a locally developed tool called TANIS [10]. The TANIS program is a shell script which configures a machine based upon the configuration description, which is distributed via NIS[1]. The description has a simple form, and includes the complete text of entries to be placed in fstab and printcap databases. TANIS is a good example of a simple tool developed by system administrators to meet local needs.

The TANIS and Prescription specifications are comparable in size. The TANIS specification consists of nearly 1200 records, with a total size of about 55K including no comments[2]. Over 900 records (around 47K) are devoted to describing individual pieces of configuration, as opposed to structure and assignment. Excluding machine names, the TANIS specification includes definitions of nearly 200 names to identify pieces of specification, and groups of services. The experimental Prescription specification contains a total of 1164 non-blank records, with 433 of those spread across 13 tables, and 597 in definitions and prescriptions. When comments are removed, the grand total drops to 934 records. The tables contain around 20K of data, and the definitions and prescriptions add around 18K including comments. The Prescription specification includes definitions of slightly more than 200 names. Note that the size values are dependent on the details of the syntactic forms.

The two specifications are largely equivalent, but they do not match exactly. Many specifications of directory attributes that are part of the TANIS description were not explicitly duplicated. Also, a few pieces of the TANIS description which are inconsistent or obsolete were not translated.

1. Network Information System.
2. The size information given here does not include machine information or machine group information for either specification.

Comparative evaluations between the two systems can also be made on a number of qualitative points. Those that tend to favour the Prescription approach are presented first, followed by those that favour the existing TANIS form.

5.3.1 Prescription Advantages

No Repetition of Detail

A TANIS description necessarily involves substantial repetition of common details, due to the fact that the complete text of table entries is included. For example, in the original description there are 112 occurrences of the string "rw,bg,intr" in filesystem definitions, and 233 occurrences of "root.wheel 755" in directory descriptions. In the Prescription model, common details are placed in prescriptions, where they do not need to be written multiple times, so each of "rw", "bg", and "intr" appear only once and the equivalent of "root.wheel 755" appears only 4 times.

Special Case Support

Special handling for servers in the Prescription case has already been mentioned. In the TANIS description, servers must be protected from importing their own services, by explicit subtractions (of which there are some 54). This places a burden on those who maintain the description, because they must remember to adjust subtractions when they make other changes. In the Prescription case, the protection problem can be solved once and for all when the prescriptions are written.

Abstraction

With TANIS, structural abstractions are mostly implicit. For example, each link required for architecture-specific namespace adjustment must be specified explicitly, although they always occur in standardized groups. The Prescription model supports abstractions more explicitly. For the case of architecture-specific links, it is possible to specify only the source location of a group in each instance, while a prescription describes the individual links for a group in general.

Consistency Maintenance

With TANIS, all consistency constraints must be manually preserved. Due to the lack of abstraction support mentioned above, common features must be duplicated for each instance of something. For example, if a decision is made to name printer spool directories a certain way, based on the printer name, an administrator must remember to do this correctly every time a printer is added. The Prescription model, however, is designed to permit common details to be encapsulated in prescriptions which change infrequently. The details of instances are specified separately. In the printer example, the spool directory name is derived from the printer name in a prescription, and printers can be added without any consideration of spool directory name at all. Thus it is easier to maintain specifications in the first place. Also, a Prescription implementation will inherently permit automated checking of instance data consistency, which TANIS does not.

Understandability

TANIS descriptions are difficult to understand due to a lack of internal documentation, the fact that organization and relationships are mostly implicit and maintained by convention, and the fact that there is minimal typing. The Prescription approach supports comments in prescriptions, tables that meaningfully organize instance data, explicit inter-table relationships, and greater typing. On the other hand, prescriptions define things in much more abstract terms than the entries in a TANIS database, and the mapping from the specification to the syntax of the system is much less transparent.

Architecture Independence

Many parts of a TANIS specification are expressed using the precise syntax of the managed system. In the Prescription model, an abstract form is used. As a result, a Prescription specification should be more portable than a TANIS one.

5.3.2 TANIS Advantages

Aggregation Convenience

In the Prescription specification, records and fields in tables are typed, so a service cannot be defined to include multiple types of things without having intermediate names for each of the pieces. TANIS has almost no concept of typing, so it is easy to define a single name which just happens to have filesystem, printer, and directory definitions associated with it. The Prescription form is more structured, and therefore is more inconvenient. On the other hand, the Prescription approach does enforce consistency.

Unconstrained Flexibility

With TANIS, the form imposes no restrictions so there is always complete flexibility. With the Prescription model, once the tables and prescriptions are established, the flexibility is strictly limited. New situations that were not envisioned when the tables were defined may require schema modification. For example, suppose that tables were originally designed based on a decision that spool directories should always have the same ownership and permissions. If the decision changed, a table structure would need to be modified, various prescriptions would need to change, and entries for all existing printers would need to be updated. With TANIS, the decision to use consistent permissions would be a convention only, and a change to accommodate a new situation would not impact older parts of the specification.

CHAPTER 6

Related Work

The problem of software configuration management has been studied in a variety of contexts. This chapter presents just a few systems which are particularly related to the work in this thesis. Almost all of the systems described here are very practical, in keeping with the focus of this thesis.

6.1 RCMS

The Raven Configuration Management System (RCMS) [4] is a system developed at UBC as part of an exploration of configuration management in general. RCMS supports management of collections of objects in the Raven [5] object-oriented system. Specifications of correct configurations are given as assertions in the first-order predicate calculus. The predicate calculus is a powerful, declarative formalism. Since the descriptive language is so powerful, it is hard for an automated system to determine what actions should be taken when the specifications are violated. A user of the RCMS must write short repair programs to accompany specifications. A collection of managed objects is monitored by RCMS, and a repair program is executed whenever the monitoring detects a violation of a specification.

The RCMS work was an major inspiration for this thesis. The basic notion of describing configurations declaratively in terms of objects comes directly from RCMS. The model presented in this thesis, however, addresses the specific problem of practical workstation configuration. Unlike RCMS, the target environment is not a uniform object-oriented distributed system, repair is entirely automated, and specifications are more highly structured, involving two parts.

6.2 Moira

The Moira system [6], from the Athena project at MIT, is directed at the problem of automating maintenance of the many pieces of data which parameterize configurations of typical workstations. Data about various services is maintained in a central database. The Moira software is capable of generating the operational files required by the various services, in the correct formats, from the central database. The system also handles distribution of files to client machines.

Moira demonstrates that the management of data does not have to be limited by the idiosyncratic formats required by operating systems. The idea of structured specifications including database tables was inspired by Moira. In Moira, however, the transfer of data from the database into various configuration files is performed by special programs. The use of prescriptions permits a more general system to be developed.

Moira provides automation with abstraction and structure, but does not help with verification and is less than transparent. It does not support much automated consistency/correctness checking. Without such checks, Moira may distribute erroneous data which prevents the system from working to deliver corrections. This problem demonstrates the value of consistency checks in specifications.

6.3 Depot

The Depot system [7] is designed to maintain third party and locally developed software in large, heterogeneous environments. The goal is integration of separately maintained packages into a common directory hierarchy without increasing dependence on central servers. Configurations may be specified in a number of ways:

1. Listing specific collections and paths to their location.
2. Providing search paths where the first instance of each collection within a path will be used.
3. Placing collections in a special directory
4. Using a combination of the above methods

Depot is capable of performing some consistency checking according to simple fixed rules based on the application. Support is also provided for moving collections of software around, which is a significant practical matter which is not addressed in this thesis. The claim is made that simple mirroring of directory hierarchies, plus simple options, are easy for both administrators and developers to understand [7, p. 157].

Depot is a good example of a tool which primarily addresses the problem of replicating a configuration on a large number of systems. Unfortunately, it is narrow in scope, with a very limited specification language. It does not provide any assistance with verification, and does not feature abstraction.

6.4 Hobgoblin

The hobgoblin [8] system is a file and directory auditor. The tool was created to automatically check conformance of systems to abstract models. The abstract model is expressed by listing files and directories and their properties. Operators are provided to state that a particular file or directory must exist, may exist, or must not exist. In addition to existence, the language permits specification of properties of files through "attribute checkers". The attributes which may be specified are mode, owner, group, size, symlink reference, and dates. The list may be expanded through addition of external checkers. There is explicit support for describing contents of directories exclusively, and nesting is supported in descriptions. Finally, there is a "delta" language, for expressing a specification as a variation of another specification. An interpreter is capable of checking systems for conformance with hobgoblin specifications.

The hobgoblin system demonstrates a practical use of declarative descriptions for verification. Unfortunately, hobgoblin has two limitations which prevent its use for more general administration. First, the specification language only handles things of one kind (files). Second, the tool is designed only for checking conformance to specification. It cannot be used to set up a system. The designers have clearly considered removing the second limitation, as they mention a notion of "enforcers" which would modify files to achieve conformance to specification.

The Prescription language is similar in descriptive power to the language used by hobgoblin, but is more general, and permits automated repair as well as verification.

6.5 Doit

The doit solution [9] is a network software management tool designed to automate the management of software configurations on large numbers of machines. Unlike hobgoblin, doit is intended to set up machines, not check them for correctness. The specification language is procedural. Three types of actions may be performed: addition of software, deletion of software, and execution of arbitrary commands. There are variants of each type of action which cause rebooting of a host after some number of steps are performed. The system uses revision levels to keep track of what has been done on a particular machine. Each action has an associated revision level. There are also special levels for actions that should be performed at the start or end of *each* run. Configurations for doit are assigned to groups of machines. The groups are declaratively specified using set logic.

The problem with a procedural form of specification is that it generally precludes any checking. Records of the state of each machine become very important in this case, and troubleshooting may be difficult. The model presented in this thesis relies on declarative specifications to simplify checking. Doit lacks the verification, abstraction and synchronism features of the model, but does have better facilities for record-keeping and software installation.

6.6 TANIS

The locally developed system called TANIS (Tagged Attribute Network Information Service) [10] was described in Chapter 5. TANIS is a lot like doit. A "service definition" can consist of a few forms of specification: description of a directory to be created, description of a symbolic link to be created, entry for a filesystem table, entry for a printer table, description of a file that should be copied, etc. Note that most of these are declarative, although no TANIS software is presently capable of checking conformance. Variables may be incorporated in specifications to achieve machine-independence.

TANIS provides flexible automation, but is limited in scope, and lacks any abstraction. It also lacks the structure for avoiding repetition, and does not provide any synchronism. A more complete comparison of the TANIS and Prescription specification forms is provided in Section 5.3.

CHAPTER 7 Conclusions

This thesis has explored a general approach to automating some aspects of configuration management, in typical distributed computing environments. The thesis has demonstrated that a general framework is suitable as the basis for automating tasks of practical importance.

Declarative description is an ideal form of input for automated management tools. Descriptions are the natural way to express configuration information. They are well suited to the tasks of verification, monitoring, and analysis. At the same time, the use of descriptions does not necessarily preclude efficient automated repair, as has been demonstrated.

7.1 Review of Contributions

The thesis presented a *model* for automating practical configuration management, a novel *language* for describing configurations, a prototype *implementation*, and a significant *experiment* that validates the work. Each of these specific contributions is reviewed here.

7.1.1 A Model of Configuration Management

A general model was proposed, based upon features of the problem. The model is practical, but is also abstract and flexible through the use of objects to model components. Its primary characteristic is reliance on declarative forms of specification for both verification and repair purposes. It features a structured specification model, in which common, structural pieces of description are written in a logic language, and varying details are expressed separately in a database.

7.1.2 The Prescription Language

The Prescription language was introduced as the vehicle for structural specification in the new model. The language provides a reasonably simple declarative formalism, carefully designed to balance the competing requirements of verification and repair. It offers modularity and other features of traditional imperative programming languages.

7.1.3 A Prototype Implementation

The implementation proved that the Prescription language *can* be implemented. Production of the implementation produced a number of insights into the challenges of automated configuration management using an object abstraction.

7.1.4 An Experiment

The experiment demonstrated the successful application of the model and language to some of the practical problems of configuring a real computing environment. The configuration of a large group of machines was described using a simple database and a set of prescriptions. The prototype implementation successfully performed automated verification and repair based on the resulting specification.

7.2 Future Work

The work in this thesis is really just a preliminary step towards the goal of general automation of configuration management. Ultimately, distributed systems should be built with intrinsic management facilities. Much more work needs to be done in order to completely identify the set of facilities which will be required. There a number of obvious avenues that could be pursued based upon the work in this thesis.

7.2.1 Addressing Open Problems

A number of the issues and problems introduced in Section 4.7 deserve further research. The proposed scheme for dealing with dependencies on implicit side-effects has not been implemented or tested, and the whole issue of implicit side-effects requires further study. The handling of atomicity and machine boundaries certainly needs work. A production implementation, based on compiler technology and featuring static analysis of specifications, would be useful.

7.2.2 Stretching the Model

The model needs to be stretched through application to different sorts of problems. One problem of particular interest is the software installation/de-installation problem. It should be possible to write prescriptions describing a configuration of a managed system in which a piece of software is properly installed. Such a set of prescriptions would ideally encapsulate all the installation information that is required, so that installation can be done (at least partially) by automated repair. It is likely that the model of this thesis is not adequate, as it stands, for solving the installation problem.

It would also be interesting to use declarative specifications to describe user environments, involving such things as the contents of environment variables.

The absence of a larger context for prescriptions was mentioned in Section 4.7.6 and deserves elaboration. Some structure is needed for groups of related prescriptions. Such a structure could incorporate means of identifying which prescriptions should be used in different situations. For instance, in a set of prescriptions describing a software package, there might be one describing what the package requires from the operating system, one describing the properly installed state, one describing the properly de-installed state, one describing the correct operational state at any time during use, and one describing the user environment required for the package. Other prescriptions might be for internal use of those described above. All would rely upon a common collection of site-specific data.

7.2.3 Improving Repair

Various approaches could be taken to increase the power and utility of automated repair. For instance, it might be useful to permit repair hints in specifications. Hints could be used to indicate which of a set of possible repair strategies is the appropriate one in a given situation. The Prescription language supports implicit hints: the choice of statement includes an implicit hint about the repair strategy to be used. Subclassing is one way of providing different repair algorithms for different situations.

Another possible way to extend automated repair would be to add a more powerful solver. This idea was discussed in Section 3.8.3. Use of a search-oriented solution engine would introduce considerable complexity and reduce efficiency, but it might be possible to limit the negative impact.

For some situations, fully automated repair is simply not feasible. Thus it may be helpful to permit repair scripts to be composed to complement declarative specifications. Adding procedural repair would be a significant adjustment to the model. Care is needed to avoid undermining the value of declarative specifications.

7.2.4 Adding Dynamic/Temporal Support

The Prescription language has no support for temporal assertions. This is one limitation which might cause problems for more dynamic configuration management situations. Automated repair in dynamic situations with temporal specifications is likely to be quite problematic.

7.2.5 Adding Consistency Checking

It is possible to specify a variety of consistency constraints using the Prescription language, and to have them checked by the prototype implementation. For instance, it is possible to express constraints on the data in tables, and have the implementation identify cases where the constraints are violated.

The thesis does not address the problem of automatic detection of inconsistencies between parts of a complete specification. For instance, it is possible to write a prescription that simultaneously requires a file to have two different permission settings. Such a prescription is unlikely to occur in practice, but there are more subtle variations of the same problem. Prescriptions from different sources might be combined to describe a system, and there might be conflicts between them. A good example of this would be the case of prescriptions describing installation of two different software packages. A configuration management system needs to be capable of detecting such conflicts automatically.

7.2.6 Improving Support for Change

The model that was presented does not adequately address the difficulties associated with variations in configuration over time. This limitation was explained in Section 4.7.7 in some detail.

As noted there, the problem of dealing with change appears to be a problem that will be very significant for automated configuration management. At a minimum, a configuration management system needs capabilities for recording what is done, and undoing it. Ideally, a system should be able to automatically determine what operations need to be performed to move from one configuration state to another, including cleanup operations.

7.2.7 Adding Security

The model that has been presented does not address security, authentication, or authorization. In practice, basic security is an important issue. A more significant research problem is the fact that in many real-world environments, the authority to determine the configuration of a machine may be shared by different people or groups. Some may have the right to control only certain aspects of system configuration.

Glossary

activated - The state of a prescription which is being processed by a configuration management system.

class - The definitional unit corresponding to a type of component. Every object is an instance of some class.

collection - A component (or an object which models one) which collects other components. For example, a directory is a collection because it collects files. The contents of collections in a managed system are an important part of the configuration state of the managed system.

component - Any distinguishable piece of a managed system, whether hardware or software.

data object - An collection of data with associated operations. This term has the meaning normally associated with the term object, in object-oriented programming.

deferred execution - The implementation scheme in which repair processing is split into two phases, separated in time. In the first phase, the sequence of operations to be performed is determined and recorded. In the second phase, the sequence is actually executed. In between the phases, the record (or do-log) may be reviewed and/or modified.

dependent attribute - An attribute, of an object, whose value is dependent on the values of other attributes, and cannot therefore be modified without impacting the values of other attributes.

descriptive adequacy - The property of a specification formalism that holds if the formalism is able to be used to describe any configuration state which the managed system may attain.

do-log - A record of operations yet to be performed.

exclusivity - The property of a specification that holds if the specification describes the configuration of a component completely, whether implicitly or explicitly.

global identifier - A value which uniquely identifies a component of a managed system (and also the object that represents it).

immutable attribute - An attribute, of an object, whose value is not modifiable during repair processing due to some property of the object and/or modelled component.

inclusivity - The property of a specification that holds if the specification describes the configuration of a component incompletely, so that multiple configurations may conform to the specification.

instance data - Data about a particular instance of something, such as a name. Also used to refer to the data stored for an object and accessible to the object implementation.

logical filesystem - An abstraction providing the appearance of a single, unified filesystem through multiple actual filesystems and symbolic links.

managed system - A collection of items, composing a system, managed by an automated tool or collection of tools. A managed system can include both hardware and software items.

master machine - The central machine that is designated to process prescriptions.

narrowed - The condition of a statement which is to be processed in verify mode, regardless of the prevailing mode. The term is also applied to prescriptions which are defined such that every activation will be narrowed.

narrowing - The processing step of restricting the mode to verify mode in the course of processing a narrowed statement.

object - An abstraction representing a component of a managed system. An object has attributes whose values represent different aspects of the configuration state of the represented component.

Object layer - The set of procedures that implement the centralized record-keeping which supports deferred execution.

object recognition - The process of determining whether or not an object already exists to represent some component.

prescription - a named, parameterized piece of specification in the Prescription language. A prescription is the unit of definition in the language, and is similar to a predicate or procedure.

repair - The process of modifying a managed system so that it has a specified configuration. Repair includes identification of discrepancies between the system and the specification, determination of a sequence of operations that can be performed to eliminate the discrepancies, and the execution of that sequence of operations.

repairable - The property of a Prescription statement that holds if the statement is suitable for automated repair.

repair mode - A Prescription processing mode in which the configuration management system will proceed with repair as discrepancies between a specification and a managed system are identified.

slave machine - Any machine, other than the master, which is part of a managed system and must be accessed during prescription processing.

truth value - A boolean value which represents the conformance of a managed system to the specification given by a Prescription statement, at a particular point in time.

undo-log - A record of operations which have been performed but which may need to be reversed.

verification - The process of comparing a configuration specification against a managed system to determine whether the managed system has the specified configuration.

verify mode - A Prescription processing mode in which no repair is attempted.

Bibliography

[1] John Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley Publishing Company, Reading Massachusetts, 1994.

[2] Lawrence A. Rowe and Brian C. Smith. Tcl-DP, online manual.

[3] Adam Sah and John Blow. A Compiler for the Tcl Language. *TCL/TK Workshop*, Berkeley, June 1993.

[4] Terry Coatta and Gerald Neufeld. Distributed Configuration Management Using Composite Objects and Constraints. *Second International Workshop on Configurable Distributed Systems*, Pittsburgh 1994.

[5] Gerald Neufeld, Don Acton, and Terry Coatta. *The Raven System.* University of British Columbia, 1992. Computer Science Technical Report TR-92-15.

[6] Mark A. Rosenstein, Daniel E. Geer, Jr., and Peter J. Levine. The Athena Service Management System. *Usenix Conference Proceedings*, Winter 1988.

[7] Wallace Collyer and Walter Wong. Depot: A tool for Managing Software Environments. *Proceedings of the Sixth System Administration Conference (LISA VI)*, page 153, October 1992.

[8] Kenneth Rich and Scott Leadley. hobgoblin: A File and Directory Auditor. *Proceedings of the fifth Large Installation Systems Administration Conference*, page 199, September 1991.

[9] Mark Fletcher. doit: A Network Software Management Tool. *Proceedings of the Sixth System Administration Conference (LISA VI)*, page 189, October 1992.

[10] Mark Majka and George Phillips. *tanis - system for boot-time machine configuration.* University of British Columbia. Computer Science online manual.

[11] Marshall T. Rose. *The Simple Book An Introduction to Management of TCP/IP based internets.* Prentice Hall, Englewood Cliffs New Jersey, 1991.

[12] ISO/IEC IS 9595. *Information Technology - Open Systems Interconnection - Common Management Information Services Definition (CMIS)*, 1991.

[13] ISO/IEC IS 9596-1. *Information Technology - Open Systems Interconnection - Common Management Information Protocol - Part 1: Specification (CMIP)*, 1991.

Appendix A
Specification Details

Most of the prescriptions produced for the specification experiment were not included in Chapter 5 in the interest of brevity. The definitions and prescriptions are included in their entirety here. The following trivial modifications have been made from the version used in the experiment:

1. Comments have been modified slightly.
2. Statements designed to produce output during processing have been removed.
3. Spacing has been modified slightly.

Apart from these differences, the code presented here is identical to that which was actually processed by the prototype implementation.

Note that the `ArchDirs` prescription was never processed (the activation in `ImportSupplemental` is commented out). This is because of the problem that the prescription depends on implicit side effects, as described in Section 4.7.5.

The contents of the various tables are an important part of the total specification. Those contents are not included in this thesis because they would not be helpful to anyone unfamiliar with the department configuration, and they would consume a lot of space. A few entries from one table are included in this appendix, to illustrate the relationship between the table definitions and the contents of the files.

A.1 Experimental Specification Definitions

```
#
# Initial Setup: load class definitions, etc.
#

import file.tcl
import fstab.tcl
import printcap.tcl

#
# Define the machines
#
set Machines [table MachineDesc machine.table | { \
   {name String} \
   {aliases List String} \
   {arch String} \
   {subnet Integer} \
} {name} {}]
set All [machine $Machines]

#
# Define tables
#
set MGroups [table MachineGroup mgroup.table | { \
   {name String} \
   {subgroups List MachineGroup} \
   {members List Machine} \
} {name} {} ]

set ServiceAssign [table ServiceAssignment assign.table | { \
   {service ServiceGroup} \
   {group MachineGroup} \
   {individual List Machine} \
} {} {} ]

set SGroups [table ServiceGroup sgroup.table | { \
   {name String} \
   {subgroups List ServiceGroup} \
   {arch String} \
   {printers List Printer} \
   {filesystems List Filesystem} \
   {logicals List FilesystemGroup} \
} {name} {{ServiceLink {name service}} \
   {ServiceFileCopy {name service}} {ServiceDir {name service}}} ]

set Exclusions [table MachineExclude exclude.table | { \
   {machine Machine} \
   {noimport List Filesystem} \
   {noimportgroup List FilesystemGroup} \
   {reject List ServiceGroup} \
} {machine} {} ]
```

```
set MachineAssign [table MachineAssignment massign.table | { \
   {machine Machine} \
   {services List ServiceGroup} \
   {filesystems List Filesystem} \
   {logicals List FilesystemGroup} \
} {machine} {} ]

set Alternates [table ServerAlternate server-alt.table | { \
   {server Machine} \
   {subnet Integer} \
   {alternate String} \
} {server subnet} {} ]

set FGroups [table FilesystemGroup fgroup.table | { \
   {name String} \
   {parts List Filesystem} \
   {root Filesystem} \
   {rootName String} \
   {rootCname String} \
} {name} {} ]

set FS [table Filesystem fs.table | { \
   {name String} \
   {server Machine} \
   {fs String} \
   {soft Boolean} \
   {quota Boolean} \
   {cname String} \
   {mountPoint String} \
   {canonicalize Boolean} \
} {name} {{FilesystemLink {name fs}} {FilesystemArch {name fs}}} ]

set FSLinks [table FilesystemLink flink.table | { \
   {fs Filesystem} \
   {source String} \
   {link String} \
} {} {} ]

set FSArch [table FilesystemArch arch.table | { \
   {fs Filesystem} \
   {source String} \
   {link String} \
   {subdirs Boolean} \
} {} {} ]

set Printers [table Printer printer.table | { \
   {name String} \
   {server Machine} \
   {aliases List String} \
   {type String} \
   {note String} \
   {maxSize Integer} \
} {name} {} ]

set SLinks [table ServiceLink slink.table | { \
   {service ServiceGroup} \
   {source String} \
   {link String} \
} {} {} ]

set FileCopies [table ServiceFileCopy copy.table | { \
   {service ServiceGroup} \
   {name String} \
   {source String} \
   {dest String} \
} {} {} ]

set Dirs [table ServiceDir dir.table | { \
   {service ServiceGroup} \
   {dir String} \
} {} {} ]

#
# Global constants
#
set root 0
set daemon 1
set wheel 0

#
# Prescriptions
#

#---------------------------------------------------
# main - describes state of entire distributed system
#
# The use of the name 'main' is arbitrary.
# The appropriate state is described in various tables.
# This prescription consists of assertions about the
# system in relation to the data in various tables

prescription main {} {
   global ServiceAssign
   forall S ServiceAssignment $ServiceAssign {} {
      # Each service is assigned to the specified group of machines
      AssignToGroup $S.group $S.service

      # Each service is assigned to the specified individual
machines
      forall M Machine $S.individual {} {
         AssignServices [Machine $M] $S.service
      }
   }
   global MachineAssign
   # Each machine has the specified individual services assigned
   forall M MachineAssignment $MachineAssign {} {
      forall S ServiceGroup $M.services {} {
         AssignServices [Machine $M.machine] $S
      }
      forall L FilesystemGroup $M.logicals {} {
         ImportGroup [Machine $M.machine] $L
      }
      forall F Filesystem $M.filesystems {} {
         GuardedImport [Machine $M.machine] $F
      }
   }
}

# ----------------------------------------------------
# AssignToGroup - Describes the assignment of
# a group of services to a group of machines
#
# Parameters:
#  mg : MachineGroup (from table)
#  sg : ServiceGroup (from table)

prescription AssignToGroup {mg sg} {

   # The group of services is assigned to machines in
   # subgroups of the group of machines
   forall M MachineGroup $mg.subgroups {} {
      AssignToGroup $M $sg
   }

   # The service group is assigned to individual machines
   forall I Machine $mg.members {} {
      AssignIndividual [Machine $I] $sg
   }
}

# ----------------------------------------------------
# AssignIndividual - Describes the assignment of a
# group of services to an individual machine
#
# Parameters:
#  m : Machine - the machine to which assignment is made
#  sg : ServiceGroup - entry in table
#
# This prescription accomodates two special cases:
# 1) A group of services may be defined for machines of a particular
# architecture only. If this is the case, the 'arch' attribute
# will contain the name of the architecture. Otherwise, the
# attribute will be empty, and the services are to be applied
# to all machines.
# 2) A group of services may not be desired on a particular machine
# even though the machine might normally qualify for the services.
# This exceptional case is described by an entry for the machine
# in the Exclusions table, with the ServiceGroup listed in the
# 'reject' attribute.

prescription AssignIndividual {m sg} {

   # When ServiceGroup is architecture specific, it only
   # applies to machines of the correct type

   if { [string length [val $sg.arch]] == 0 || \
      [strEQ $sg.arch $m.arch] } {

      global Exclusions
      if { [in -key $m.name $Exclusions] } {
         forall E MachineExclude $Exclusions \
           {[strEQ $E.machine.name $m.name] && ! [in $sg $E.reject]}
{
           AssignServices $m $sg
         }
      } else {
         AssignServices $m $sg
      }
   }
}

# --------------------------------------------------
# AssignServices - describes the assignment of the services
# that are part of a service group, to a particular machine
#
# Parameters:
#  m : Machine
#  sg : ServiceGroup (from table)
#
# Individual services which compose a service group are listed in
# attributes of the ServiceGroup, by type. For instance, there is
# an attribute 'printers' whose value is the list of printer services
# that are part of the group. The group is actually defined in a
table,
# where the value of an attribute like 'printers' is a list of keys
of
# records describing printer services.
#
# For some service group requirements, the approach described above
# does not work well, because the requirement is not a separated
service
# which could be independently assigned. For these cases, separate
# tables describe the requirement, by service group name.

prescription AssignServices {m sg} {
   # Subgroups of the ServiceGroup must be assigned
   forall G ServiceGroup $sg.subgroups {} {
      AssignIndividual $m $G
   }

   # Printers
   forall P Printer $sg.printers {} {
      PrinterDef $m $P
   }

   # Filesystems
   forall F Filesystem $sg.filesystems {} {
      GuardedImport $m $F
   }

   # Logical Filesystems
   forall F FilesystemGroup $sg.logicals {} {
      ImportGroup $m $F
   }

   # Symbolic Links associated with the service group must exist
   global SLinks
      forall L ServiceLink $SLinks { [objEQ $L.service $sg] } {
      SymLink $m $L.source $L.link
   }

   # Arbitrary directories associated with the service group must
exist
   global Dirs
   forall D ServiceDir $Dirs { [objEQ $D.service $sg] } {
      dir $m $D.dir
   }


   # File copies required for the service group must exist
   global FileCopies
   forall F ServiceFileCopy $FileCopies { [objEQ $F.service $sg] } {
      FileCopy $m $F.name $F.source $F.dest
   }
}

# ---------------------------------------------------
# FileCopy - describes a file that is required to exist
# on a machine, and to have the same contents as a file
# of the same name in another directory on the machine.
#
# Parameters:
#  m : Machine - machine on which file must exist
#  name : name of file
#  source : name of dir in which source file exists
#  dest : name of dir in which copy is required to exist

prescription FileCopy {m name source dest} {
   # Clearly the source file must exist
   narrow { require S File $source/$name -closure $m.root {} }

   # The directory for the copy must exist
   global root daemon
   dir $m $dest $root $daemon 0755

   # The copy file itself must exist
   require S File $source/$name -closure $m.root {
      require D File $dest/$name -closure $m.root {
         logical $D.contents eq $S.contents
         logical $D.perms == $S.perms
         logical $D.owner == $S.owner
         logical $D.group == $S.group
      }
   }
}

# ---------------------------------------------------
# SymLink - describes a symbolic link that must exist
#
# Parameters:
#  m : Machine on which link must exist
#  source : name to which link points
#  dest : name of link

prescription SymLink {m source dest} {
   require L SymLink $dest -closure $m.root {
      logical $L.ref s= $source
   }
}

# ---------------------------------------------------
# dir - describes a directory that must exist
#
# Parameters:
#  m : Machine on which dir must exist
#  path : String pathname of directory
#  owner : Integer owner uid
#  group : Integer group gid
#  mode : Integer permission mode
prescription dir {m path owner group mode} {
   require D Directory $path -closure $m.root {
      logical $D.owner == $owner
      logical $D.perms == $mode
      logical $D.group == $group
   }
}

#---------------------------------------------------
# PrinterDef - describes a remote printer definition that
# must be present on a machine
#
# Parameters:
```

```
#  m : Machine
#  pd : Printer (from table)
#

prescription PrinterDef {m pd} {
   global All
   narrow { or {
      require M Machine [val $pd.server.name] $All {}
      require M MachineProxy [val $pd.server.name] $All {}
      }
   }

   if { [strEQ $pd.server.name $m.name] } {
      # Printer should be locally defined on its server
   } else {
      # A PrinterEntry must exist
      require P PrinterRecord [val $pd.name] $m.printcap {
         logical $P.name s= $pd.name
         logical $P.aliases s= $pd.aliases
         logical $P.lp s= ""
         logical $P.rm s= $pd.@server
         logical $P.rp s= $pd.name
         if { [strEQ $pd.maxSize ""] } {
            logical $P.mx == 0
         } else {
            logical $P.mx == $pd.maxSize
         }
         logical $P.sd s= /usr/spool/print/[val $pd.name]
         logical $P.ty s= $pd.type
         logical $P.note s= $pd.note

         # The specified spool directory must exist
         global root daemon
         dir $m $P.sd $daemon $daemon 02755
      }
   }
}

# --------------------------------------------------
# ImportGroup - describes the importing of a group of
# filesystems by an individual machine
#
# Parameters:
#  m : Machine
#  fg : FilesystemGroup (from table)
#

prescription ImportGroup {m fg} {
   # Each part must be imported individually, if not explicitly
   # excepted for this machine
   global Exclusions
   if { [in -key $m.name $Exclusions] } {
      forall E MachineExclude $Exclusions {[objEQ $E.machine $m]} {
         if { ! [in $fg $E.noimportgroup] } {
            SafeImportGroup $m $fg
         }
      }
   } else {
      SafeImportGroup $m $fg
   }
}

# ---------------------------------------------------
# SafeImportGroup - describes the importing of a group
# of filesystems by an individual machine from which
# they are not excluded.
#
# Parameters:
#  m : Machine
#  fg : FilesystemGroup (from table)
#

prescription SafeImportGroup {m fg} {

   # Every filesystem in the group must be imported
   forall F Filesystem $fg.parts {} {
      GuardedImport $m $F
   }

   # Master root dir must exist if specified
   if { ! [isNIL $fg.root] } {
      if { [strEQ $fg.root.server.name $m.name]} {
         # This machine is server for the filesystem containing root
         # dir
         if { [val $fg.root.canonicalize] } {
            # Server name space should be adjusted
            SymLink $m [val $fg.root.fs]/[val $fg.rootName] \
              $fg.rootCname
         }
      } else {
         # This machine is a regular client
         if { [strNE $fg.root.mountPoint ""] } {
            SymLink $m [val $fg.root.mountPoint]/[val $fg.rootName] \
              $fg.rootCname
         } else {
            SymLink $m /nfs/[val $fg.root.name]/[val $fg.rootName] \
              $fg.rootCname
         }
      }
   }
}

# ---------------------------------------------------
# GuardedImport - describes import of a single filesystem
# on a single machine if not excluded.
#
# Parameters:
#  m : Machine
#  f : Filesystem (from table)

prescription GuardedImport {m f} {
   global Exclusions
   if { [in -key $m.name $Exclusions] } {
      forall E MachineExclude $Exclusions { [objEQ $E.machine $m] }
{
         if { ! [in $f $E.noimport] } {
            ImportSingle $m $f
         }
      }
   } else {
      ImportSingle $m $f
   }
}

# ------------------------------------------------
# ImportSingle - describes import of a single filesystem
# on a single machine.
#
# Parameters:
#  m : Machine
#  f : Filesystem (from table)
#

prescription ImportSingle {m f} {
   if { [strEQ $f.server.name $m.name] } {
      # Importing is not done on server, but namespace adjustment
      # may be
      if { [val $f.canonicalize] } {
         ImportSupplemental $m $f $f.fs 1
      }
   } else {
      if { [strNE $f.mountPoint ""] } {
         NFSImport $m $f.@server $f.fs $f.mountPoint $f.soft
$f.quota
         ImportSupplemental $m $f $f.mountPoint 0
      } else {
         # Mounts below /nfs
         NFSImport $m $f.@server $f.fs /nfs/[val $f.name] $f.soft \\
           $f.quota
         ImportSupplemental $m $f /nfs/[val $f.name] 0
      }
   }
}

# ------------------------------------------------
# ImportSupplemental - describes the state associated
# with import of a filesystem
#
# Parameters:
#  m : Machine
#  f : Filesystem (from table)
#  mount : String name of mount point for filesystem
#  server : Integer - 1 if m is server, 0 otherwise
#
# The supplemental state involves various symbolic links.


prescription ImportSupplemental {m f mount server} {
   # Symlink for name space required
   # if cname is "", this link is not required
   if { [strNE $f.cname ""] && \
         ( ! $server || [strNE $f.cname $f.fs] ) } {
      SymLink $m $mount $f.cname
   }

   # Other symbolic links
   global FSLinks
   forall L FilesystemLink $FSLinks { [objEQ $L.fs $f] } {
      SymLink $m $mount/[val $L.source] $L.link
   }

   # Architecture specific setup: the generic, bin, lib dirs
   global FSArch
   forall L FilesystemArch $FSArch { [objEQ $L.fs $f] } {
#     ArchDirs $m $mount $L
   }
}

# --------------------------------------------------
# NFSImport - describes import of a filesystem via NFS
#
# Parameters:
#  m : Machine
#  server : Machine that is server server
#  fs : String name of filesystem on server
#  mountPoint : String name of mount point
#  soft : Integer - 1 if soft mount required
#  quota : Integer - 1 if quotas must be enabled
#

prescription NFSImport {m serverName fs mountPoint soft quota} {
   global All
   narrow { logical {} {} [in -key $serverName $All] }

   # Mount point directory required
   global root wheel
   dir $m $mountPoint $root $wheel 0755

   # Handle possibility of alternate servers for particular subnets
   global Alternates
   if { [in -key [list [val $serverName] [val $m.subnet]] \
      $Alternates]} {
      require A ServerAlternate \
         [list [val $serverName] [val $m.subnet]] $Alternates {
         fs $m $A.alternate $fs $mountPoint $soft $quota
      }
   } else {
      fs $m [val $serverName] $fs $mountPoint $soft $quota
   }
}

# --------------------------------------------------
# fs - describes a filesystem that must be defined on
# a machine
#
# Parameters:
#  m : Machine
#  server : String name of server
#  fs : String name of filesystem on server
#  mountPoint : String mount point
#  soft : Integer - 1 if soft mount required
#  quota : Integer - 1 if quotas must be enabled
#

prescription fs {m server fs mountPoint soft quota} {
   narrow { logical {} {} ![in $server $m.aliases] }

   require F FileSysRecord [list $server $fs] $m.fstab {
      # Same regardless of hard or soft mount:
      logical $F.dir s= $mountPoint
      logical $F.type s= nfs
      logical $F.options contains rw
      logical $F.options contains intr

      # Vary by mount type:
      if { $soft } {
         logical $F.options contains retry=2
         logical $F.options contains timeo=20
      } else {
         logical $F.options contains bg
      }

      if {! $quota} {
         logical $F.options contains noquota
      }
   }
}

# --------------------------------------------------
# ArchDirs - Describe symlinks that must exist for architecture
# specific directories.
```

```
#
# Parameters:
#  m : Machine
#  root : String root of architecture specific dirs
#  a : FilesystemArch (from table)

prescription ArchDirs {m root a} {
   set path $root/[val $a.source]
   forall D Directory -closure $m.root \
      { [globEQ "$path" $D.fullPath] } {
      if { [in -key $m.arch $D.contents] } {
         # Architecture specific dirs are here: links are
appropriate
         if { [val $a.subdirs] } {
            # Link specific subdirs below the arch-specific dir
            global root daemon
            dir $m $a.link $root $daemon 0755
            SymLink $m [val $D.fullPath]/[val $m.arch]/bin \
              [val $a.link]/bin
            SymLink $m [val $D.fullPath]/[val $m.arch]/lib \
              [val $a.link]/lib

            set path [val $m.arch]
            forall D2 Directory $D { [globEQ "$path" $D2.fullPath] }
{
              if { [in -key obj $D2] } {
                SymLink $m [val $D.fullPath]/[val $m.arch]/obj \
                  [val $a.link]/obj
              }
            }

            if { [in -key share $D] } {
              SymLink $m [val $D.fullPath]/share [val $a.link/share]
            } else {
              SymLink $m [val $D.fullPath]/generic \
                [val $a.link]/generic
            }
         } else {
            # Link the arch-specific dir itself
            SymLink $m $D.fullPath $a.link
         }
      }
   }
}
```

A.2 Sample Table Entries

Here are a few sample entries from the `Printer` table, as they appear in the file `printer.table`:

```
hp306|hp306|{HP 4si (in CC306)} *c-hp4si,CC306|HP 4si|Room CC306|
lw106|lw106|garibaldi laserwriter {NEC Silentwriter (in CC106) 1} \
*c-nec,CC106|NEC SilentWriter|Room 106|
cicsrlw|lwcicsr|lp lwc default {Silentwriter (in CC289)} \
*c-nec,CC289|||10000
lw238|lw238|clinker clink *c-nec,CC238|NEC SilentWriter|Draft Room\
312|
```

In this case, backslash characters are inserted to indicate where a single line of the file has been broken for presentation. The backslashes do not actually appear in the file.