Prescriptions:
A Language for Describing Software Configurations

by
Jim Thornton
thornton@cs.ubc.ca

Technical Report 94-18
June 1994

Abstract

Automation of software configuration management is an important practical problem. Any automated tool must work from some specifications of correct or desired configurations. This report introduces a language for describing acceptable configurations of common systems, so that automated management is possible. The proposed language is declarative, while at the same time there are efficient algorithms for modifying a system to conform to a specification in many cases of practical importance.

Department of Computer Science
University of British Columbia
2366 Main Mall,
Vancouver, BC
CANADA V6T 1Z4

1 Introduction

Correct operation of computing systems depends upon correct configuration. Software is an essential part of any computing system, and contributes a great deal of complexity to the problem of configuration management. A typical computing system contains many instances of different software constructs such as files, directories, processes, tables, ports, sockets, mailboxes, queues, threads, segments, programs, drivers, etc. Each type of construct has a variety of states it can attain. Thus the software state space of a system as a whole is immense. Software states are also subject to frequent and rapid change. A significant amount of knowledge is frequently required to understand the relationships between components. Furthermore, software systems are notoriously *fragile*. Minor errors in configuration can seriously interfere with operation.

The problem is even worse in distributed systems, since there are more items to manage. When a distributed system is built by linking many autonomous systems, as is common in practice today, substantial duplication of configurations is generally required.

Due to the size of the problem alone, it is no longer possible for humans to manually manipulate software components to achieve acceptable system states. Automation has become essential.

Any automated management system must support some means of specifying desirable or acceptable configuration states. The specification formalism is of critical importance in determining what the configuration management system will be able to do automatically, as well as the ease with which features will be accessible. For these reasons, the nature of the specification language is fundamental to the success or failure of a practical configuration management system.

This report introduces a language for describing configurations of software in distributed computing systems. It is a language designed to balance competing objectives and to be of practical use. The sample application is the management of large collections of autonomous workstations and servers running UNIX-like operating systems. The report serves as a preliminary snap-shot of work in progress.

The body of the report begins with a review of some related work that is of interest. An introduction is then provided to the problem of describing configurations, and the resulting design tension. The language is introduced and examples provided. Finally the syntax and semantics are presented. The report concludes with a section on future work. The appendices provide a grammar for the language, and more example material.

2 Related Work

The problem of software configuration management has attracted attention in a variety of contexts. As a result, the work that has been done is quite varied. The purpose of this section is not to provide a complete survey, but rather to highlight a few developments which are particularly related to the work in this report.

The Raven Configuration Management System (RCMS) [CN94] 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 [ACN92] object-oriented system. Specifications of correct configurations are given as assertions in first-order predicate calculus. The predicate calculus is a powerful, declarative formalism. Since the descriptive language is so powerful, it is very 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 important inspiration for the design of the language described in this report.

The Moira system [RGL88], from the Athena project at MIT, is directed at automated maintenance of 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 handles only a part of the configuration problem, but it is very interesting for a couple of reasons. First, the usefulness of Moira demonstrates the importance of descriptive data in common software configurations. Secondly, Moira is an example of how the management of data does not have to be linked to the idiosyncratic formats so often required by software. Data can be maintained in a unified database which suits the needs of administrators, then translated automatically to the formats required by software. Moira does not support much automated consistency/correctness checking. Without such checks, Moira can distribute erroneous data which prevents the system from working to deliver corrections. This problem demonstrates the value of consistency checks in specifications.

The Depot system [CW92] 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 hierarchy without increasing dependence on central servers. Configurations may be specified in a number of ways:

- Listing specific collections and paths to their location.
- Providing search paths where the first instance of each collection within a path will be used
- Placing collections in a special directory
- Using a combination of 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 certainly and important practical task. The claim is made that simple mirroring of directory hierarchies plus simple options are easy for both administrators and developers to understand [CW92, p. 157] Depot is a good example of a tool which primarily addresses the problem of duplicating a configuration on a large number of systems. Unfortunately, it is narrow in scope, with a very limited specification language.

The `hobgoblin` [RL91] 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, dates, and 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. This is an example of a practical use of declarative specifications. Unfortunately, hobgoblin has two limitations which prevent its use for more general administration. First, the specification language only handles things of one kind (files). Secondly, 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 `doit` solution [Fle92] 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 therefore procedural. Three types of actions may be performed: adding software, deleting software, executing arbitrary commands. There are variants of each type of action which cause rebooting of a host some number of actions are performed. The system uses revision levels to identify 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.

A locally developed system called TANIS (Tagged Attribute Network Information Service) [MP] is used to manage machines in the computer science department at UBC. 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.

3 Describing Configurations

In order to build *general* tools for automated management of software configurations, there must be a *general* way of describing configurations. Why promote general solutions? There are a few reasons:

- The problems are pervasive, not restricted to a limited area.
- Each separate solution imposes a learning burden on administrators.
- As a matter of architecture, I believe that systems must eventually become self-describing in order to support high levels of automation in management. In order to achieve such a situation, a general specification mechanism may be helpful.

It is simple to write descriptions of configurations in natural language. As an example, consider the configuration of a shared filesystem. Under Unix with NFS, an informal description of the valid configuration might look something like the following:

> *Correct shared filesystem configuration*
> On the server, there must be an entry for the filesystem in the /etc/exports database. The directory field should contain the name of the directory where the filesystem is locally mounted. The clients list should have an entry for each client, with the name of the client. On each client, there must be a directory /*name*, where *name* is the name of the shared filesystem. The directory must be owned by `root.daemon` and have mode `0777`. An entry is required in the /etc/fstab database. The node field should contain the name of the server, and the filesystem field should contain the name of the directory where the filesystem is locally mounted on the server. The name of the shared filesystem must be in the directory field of the fstab database entry. The type is always `nfs`, options should include the proper access string plus `bg intr`, and the last two fields should both have the value `0`.

The power of natural language is certainly adequate, but there is a price – we cannot write software to efficiently use natural language descriptions. We need a formalism that sacrifices some power in order to achieve practicality. This requirement implies a delicate balancing act.

Since the limiting factor is the practical usability of the specifications, we need to consider anticipated uses. There are two fundamental operations which will be performed with specifications:

Verification: Checking the distributed system for conformance to specification.

Repair: Modifying the distributed system so that it conforms to a specification.

For verification, a powerful, declarative formalism like the predicate calculus is ideal. Unfortunately, repair is easiest if there is a simple procedural language. The obvious solution when confronted by this tension is to create a hybrid solution: declarative assertions used for verification, and programs used for repair. This is exactly the approach taken in RCMS [CN94].

The central motivating assertion of this report is that a single form of specification is preferable to a hybrid approach. The language must be declarative, but at the same time there must be simple and efficient algorithms for modifying systems to achieve conformance.

The fundamental realization is that declarative simplicity must be sacrificed to achieve efficient repair. It will be appropriate in some cases to have automated repair, while in other cases the descriptive power will be more important. The language must support both situations.

4 A Language of Prescriptions

The language of prescriptions is a language for writing modular descriptions of configurations of distributed systems. The basic unit of description is called a *prescription*, and is similar in purpose to a procedure in a traditional programming language, or a predicate in a logic programming language. All statements are contained within some prescription. Prescriptions themselves are found within a larger context, but that is not the subject of this paper.

Prescriptions are intended for automated processing as described in the preceding section.

The language is designed to combine declarative power with certain limitations necessary to permit automated repair. Here are a few specific objectives:

1. The language must support *both* specifications suitable for automatic repair and those which are not. The descriptive power should not be limited when automated repair is not required.
2. Prescriptions should have a straightforward meaning, both in declarative and procedural senses. Specifications should be easily understood by those who must read them.
3. There must be a syntactic distinction between descriptions suitable for automated repair, and others.
4. There must be a syntactic distinction between descriptions intended for repair and others.

In order to describe configurations, it is necessary to have some way of referring to states of individual items. In the language of prescriptions, an *object model* is used. Every type of item is represented as an object with a set of typed attributes. A single syntax is used to identify attributes of all objects, regardless of the actual form of the item represented. Most objects directly model software constructs that must be managed. Management data, however, can also be referenced in the same way. For example, a filesystem can be described by a data record whose fields are addressed as attributes of an object.

In a typical object-oriented language, objects have methods as well as attributes. Since prescriptions express desired states declaratively, methods on objects are not represented in the language.[1] Note that traditional inheritance is useful for handling classification hierarchies. For instance, a plain file in UNIX can be considered as one type of file.

Details of data definition, object definition, and typing are beyond the scope of this report.

By way of illustration, consider the following sample prescription:

```
prescription correctMode(d: Directory, mode: Integer)
{
  foreach F: File in d.contents {
         (
            F.mode = mode
```

[1] For implementation purposes of course, methods on objects are required. The point is that methods do not appear in the language.

```
        )
  }
}
```

In natural language, an equivalent description is the following:

> **Given:** A directory d, and a mode *mode*: The *correctMode* configuration holds when every file in d has mode *mode*

Verification processing is straightforward because the description is declarative. In this example, repair is also straightforward, because enough information is provided. The objects to which repair operations might be applied are clearly identified, and the corrective action required when the specification is violated is simple.

4.1 An Extended Example

The extended example presented here is intended to illustrate the use of the language of prescriptions in a practical context, so that the subsequent explanation of language details will be easier to place in context.

The example is set in the context of a hypothetical company which will be called Hedgehog Inc. Hedgehog uses Unix workstations in every area of operations. All the workstations are connected in a TCP/IP internet, and all are capable of mounting or exporting file systems with NFS, and accessing remote printers via the lpd protocol.

Assume that the company is using a configuration management system (CMS) based on the language of prescriptions. Many details of the operation of the system are beyond the scope of this example, and in fact, the entire report. A few assumptions must be made, however. The configuration management system is distributed, with pieces that run on each workstation. The CMS comes with object definitions for the various constructs in Unix systems, like files and processes. Entries in typical data files, like the password file, also have object representations. The CMS permits administrators to define tables which are not part of a standard Unix system. The entries in these tables may be referenced as objects. Thus all of the variable data which parameterizes configurations can be managed in a simple database. The CMS allows global identifiers to be created and bound to particular objects or tables. The CMS also accepts any number of prescriptions which together describe the acceptable state of the entire system.

Hedgehog machines are organized by corporate division. The four divisions are administration, manufacturing, product development, and sales. A few machines are designated as corporate machines because they serve all divisions.

This example illustrates how filesystem sharing with NFS might be described. Most machines need to mount filesystems on corporate servers. In addition, machines in each division generally mount filesystems on division servers. Filesystems are managed in groups by logical purpose. If one filesystem in a group is imported to a particular system, all the filesystems in the group must be imported to that system. Such a group of filesystems is called a *logical filesystem*.

The Hedgehog administrators define a few tables to contain variable information. In this presentation of the tables, the data type associated with each attribute is given under the name of the attribute. The base types used in this example are "String", which is an ordinary sequence of characters, and "LIST of *type*", which is an ordinary list aggregate type. The italicized type names (*Machine*, *Filesystem*, *Netgroup*, *Logical*) indicate records from other tables represented by a foreign key. Also note that only a few sample records are presented for each table.

Netgroup Table	
name **key** String	**members** LIST of *Machine*
administration	white.hh.com, red.hh.com, blue.hh.com, ...
development	magellan.hh.com, enterprise.hh.com, ...
corporate	gilbert.hh.com, sullivan.hh.com, huey.hh.com, ...
sales	tulip.hh.com, daffodil.hh.com, ...

The **Netgroup** table defines groups of machines. An sample **Machine** table is not presented here.

Logical Table		
name	**parts**	**root**
key String	LIST of *Filesystem*	*Filesystem*
admin	admin1, admin2, admin-user	admin1
apps	licensed, public, source	public
corporate	corp-users, support	support

The **Logical** table defines logical filesystems in terms of the physical filesystems they contain. The **root** attribute identifies a particular filesystem from the **parts** list as one containing a "root" directory for the entire logical filesystem.

Filesystem Table		
name	**server**	**fsname**
key String	*Machine*	String
admin1	gilbert.hh.com	/disk0
corp-users	gilbert.hh.com	/disk1
support	sullivan.hh.com	/disk2
licensed	gilbert.hh.com	/fs/software
public	sullivan.hh.com	/fs/public

The **Filesystem** table provides information about individual physical filesystems.

Mounts Table		
fs	**group**	**access**
Netgroup	*Filesystem*	String
corporate	corporate	rw
corporate	administration	r
corporate	development	r
corporate	sales	r
corporate	manufacturing	r

The **Mounts** table links logical filesystems to groups of machines. The **access** attribute is a string specifying the type of access that machines in the group should have to the logical filesystem.

A number of prescriptions are used to describe parts of filesystem configuration, using parameter values that ultimately come from the tables. For example, the following prescription describes the mounting of a single filesystem with NFS:

```
prescription mountsNFS(m: Machine, f: Filesystem,
                       access: String)
{
  require D: Directory <fullPath = "/net/"+f.name> in m.files {
        (-- Standard perms for mount points
            D.mode = 0755
        )
  }
  require F: FileSysEntry <server = f.server.name,
                           name = f.fsname>
        in m.fileSysTable {
        (
            F.mountAt = "/net/"+f.name,
            F.type = "nfs",
            F.options contains access
        )
  }
}
```

Even without a complete explanation of the language, it is not difficult to understand the meaning of the `mountsNFS` prescription. It specifies that there must be a directory and a filesystem entry on the machine m. The directory is identified as /net/*name*, where *name* is the name of filesystem f. The directory must have the mode `0755`. The filesystem entry, in the machine's table, is identified by server and name. The required contents of three other fields in that entry are specified. The double hyphen (`--`) token marks the remainder of the line as a comment.

Since logical filesystems are important units, there is a prescription which describes the configuration of a machine importing a logical filesystem:

```
prescription mountsLogical(logFS: Logical, machine: Machine,
                           access: String)
{
  -- Each component physical filesystem is mounted
  foreach F: Filesystem in logFS.parts {
```

```
    if (F.server != machine) {
        -- Export and import via NFS
        mountsNFS(machine, F, access)
        exportsNFS(F.server, F, machine, access)
    } else {
        -- Create local sym links to canonicalize
        -- namespace on server
        localLinks(machine, F)
    }

    -- Link to master directory
    fsLinks(machine, logFS)
  }
}
```

The `mountsLogical` prescription relies on a number of other prescriptions. The first of these is the `mountsNFS` prescription given above. The `exportsNFS` prescription from the preceding section is also used. Definitions of `localLinks` and `fsLinks` follow.

```
prescription localLinks(m: Machine, f: Filesystem)
{
  require S: SymLink <fullPath = "/net/"+f.name> in m.files {
         (
            S.reference = f.fsname
            -- f.fsname is assumed to be absolute pathname to
            -- mount point of physical filesystem
        )
  }
}

prescription fsLinks(m: Machine, logFS: Logical)
{
  -- Make link /{name} -> /net/{root}/dir
  -- where {name} is name of logical FS
  --       {root} is name of physical fs containing root dir
```

```
  -- By convention, root dir for logical FS is /dir on one
  -- of physical filesystems

  require SymLink <fullPath = "/"+logFS.name> in m.files {
         (
            reference = "/net/"+logFS.root.name+"/dir"
        )
  }
}
```

Finally, prescriptions are needed to link those given above to the data in the various tables. Every record in the **Mounts** table indicates that a particular logical filesystem should be mounted on a particular set of machines. The following prescription expresses that idea. Assume that the list of records in the **Mounts** table are bound to the global identifier `$mounts`.

```
prescription filesystems()
{
  foreach M: MountEntry in $mounts {
    GroupMountLogical(M.fs, M.group, M.access)
  }
}
```

The `GroupMountLogical` prescription describes the mounting of a logical filesystem on all the machines in a particular group.

```
prescription GroupMountLogical(logFS: Logical,
                               group: Netgroup, access: String)
{
  -- Each machine in the group mounts the logical filesystem
  foreach M: Machine in group.members {
        mountsLogical(logFS, M, access)
  }
}
```

The `mountsLogical` prescription has already been defined.

This completes the extended example of filesystem configuration at Hedgehog Inc. There are a variety of consistency constraints which could be associated with the various prescriptions. The language is capable of expressing a number of such constraints, but to include them at this point would require too much explanation. More example material is provided in appendix B.

5 Syntax

This section describes the syntactic elements of the language. The meaning of the different pieces of syntax is explained later.

In the syntax illustrations, **bold face** type is used for tokens that must appear exactly as shown, while *italic* type is used for placeholders that must be replaced by specific values dependent on situation. Optional parts are enclosed in square brackets (`[ ]`).

5.1 Lexical Form

Lexically, the language is like many typical programming languages, such as Pascal or C. A string in the language consists of a series of tokens. As in C, tokens are identifiers, keywords, constants of various types, string literals (really a special form of constant), operators, or other separators[KR88, p. 191]. Also like C, white space characters are significant only as token separators unless they occur in string literals. Most tokens must be separated from other tokens by white space. The exceptions are operators and delimiters. Note that string literals may contain any white space characters, including newlines. No continuation character is required.

Comments begin with two hyphens and always end at the next line break. Comments may alternatively be terminated before a line break by another pair of hyphens. Comments do not nest.

5.1.1 Identifiers

Local identifiers are composed of alphabetic characters, digits, the underscore, and the hyphen. Identifiers must begin with an alphabetic character.

Global identifiers are local identifiers with a dollar sign ($) at the beginning.

5.1.2 Keywords

There are a number of keywords in the language:
prescription, **narrow**, **foreach**, **with**, **if**, **require**, **disallow**, **in**.

5.1.3 Constants

Constants may be written for any of the basic types. The common integer and character constants have familiar forms. More complex constants are not discussed in this report.

There are a few special values, which are represented by tokens consisting of a pound sign (#) followed by a number of uppercase characters. The special values are **#NIL**, and **#ANY**.

5.1.4 String Literals

String literals consist of a series of characters delimited by double quotation characters ("). The escape character mechanism of C is supported.

5.1.5 Operators

The language provides a relatively large number of operators. Operators are often composed of special symbols, but are sometimes complete words. Many common operators are borrowed from C. The list here is not necessarily complete.

Class	**Sample Operators**
Logical Binary	&& , $\vert\vert$
Logical Unary	!
Relational	= != < > <= >= =< => **contains** **in**
Computational	+ − * /

5.1.6 Separators

There are a variety of separators:
( ) [ ] { } < > , : {| |}

5.2 Prescription Definitions

The syntax of a prescription definition is shown below:

prescription [**narrow**] *name* (*param-def* [, ...]) *block*

Like a procedure, a prescription has a name and an argument list in which formal parameters are declared with names and types. The body of one prescription may contain a statement activating another prescription, supplying values for each parameter. This activation statement is similar to procedure call. Recursion is permitted.

The optional keyword **narrow** indicates that the entire prescription is narrowed whenever it is activated. Narrowing is explained later.

5.3 Blocks

A block is a scope unit containing statements, just as in common procedural languages. Unlike other languages, there are two types of blocks[2], distinguished by their delimiting tokens. The two block types have the following names and syntax:

AND-block:
{ *statement* [...]}

OR-block:
{| *statement* [...]|}

The meaning of these blocks will be described in the section on semantics. Blocks have the same syntax regardless of the context in which they appear.

Blocks may be nested inside other blocks in two ways. First, a block is itself a legal statement, and may therefore be directly nested inside another block. More commonly, blocks appear as part of statements. This is the case with the block in the **foreach** statement in the example in section 4. Wherever a block is required, either type may be supplied.

The language does not support traditional declaration of local or automatic variables. Thus direct nesting of one block inside another does not serve to open a new identifier scope, as it might in a regular programming language. The reason for nesting a block directly inside another is related to

[2]A third type was considered but it has been rejected because of repair difficulties.

the *meaning* of the different types of blocks.

5.4 Statements

Prescriptions contain a series of statements inside a block. There are a few different statement forms:

Block: As explained earlier, blocks are themselves legal statements.

Prescription: Prescription activation has a syntax like that of function call in C. Note that prescriptions do *not* return a value.

Regular: Regular statements are composed of keywords, blocks, logicals, identifier declarations, etc. separated by white space. Every regular statement starts with an identifying keyword. The precise syntax varies by statement. In the example prescription in section 4, the **require** statement is an example of a regular statement consisting of the identifying keyword **require**, an identifier declaration (for `E`), an object reference, the keyword **in**, a data reference, and a block.

Logical: Logical expressions are legal statements. They are described in the next section.

Narrow: A narrowed statement consists of square brackets around any other legal statement. Square brackets may be nested, but the effect is the same regardless of the level of nesting. Narrowing has to do with the processing mode and will be explained later.

Note that narrowing of the entire body of a prescription is indicated by adding the **narrow** keyword to the prescription definition, *not* by adding square brackets around the main block, since a prescription definition must contain a block and not just any statement.

Statements are not separated or terminated by any special token. It is not possible to select a single token that represents the relationship between statements in all cases, since that relationship varies with the type of the containing block.

5.5 Logicals

Logical expressions are used in certain contexts. They have the same syntax regardless of where they appear:

(*logical-expression*)

The logical expression is composed of relations and standard boolean operators. A relation is an expression involving a relational operator and various identifiers, constants, arithmetic operators, etc. Here is an example logical containing one relation:

```
(F.server = machine+".cs")
```

Note that there is a simple value on the left side of the operator (=) in this example. When there is only a simple value on the left side of every relational operator, the logical is in repairable form.

While parentheses are always used to delimit logicals, not everything delimited by parentheses is a logical. For example, parameter lists are delimited by parentheses. Note also that logicals may include parenthesized sub-expressions.

5.5.1 Regular Statements

Here is the syntax of each regular statement:

foreach *decl* **in** *id* [**with** *logical*]
require *decl* [*ref*]**in** *collection block*
disallow *decl* **in** *id* [**with** *logical*] *block*
if *logical block* [**else** *block*] *block*

5.6 Identifier Declarations

Declaration of local identifiers occurs in only two contexts. The first is in the formal parameter list of a prescription, and the second is in regular statements. Note that local identifiers may *not* be declared at the start of a block. There is no reason to do this, since there is no explicit way to assign a value to a local identifier. Identifiers only become bound through prescription activation (in the case of formal parameters) or implicitly through a statement.

Declarations follow the Pascal style in which the new identifier(s) appear prior to a colon (:), which is then followed by a type name.

Declaration of global identifiers, and binding of those identifiers to values, is done in the larger context outside individual prescriptions.

5.7 Data References

The language supports references to data. An object-oriented data model is used for representing managed components and organizational databases alike. Thus the majority of data references have the form of object attribute references and the syntax used for structure field reference in Pascal. These data references involve identifiers, most commonly local identifiers.

A few special symbols were introduced in section 5.1.3. These symbols represent special values. Here are the special values presently defined:

Identifier	Description
#NIL	The non-value
#ANY	Wildcard matching any value

5.8 Object References

Some statements include a reference to a particular object which may exist in a collection of objects. The reference is given by supplying values for one or more attributes. The attributes selected *must* comprise a candidate key for objects in the collection.

These references have a special syntax:

< [*attrib*=]*expression* [, ...]>

For example:

```
<fullPath = "/net/"+f.name>
```

The candidate key does not have to be minimal. Thus a reference could include a value for *every* attribute. Also note that the attribute name does not have to be specified, for those cases when there is a standard key involving only a single attribute.

6 Semantics

This section explains the concepts of the language and the meaning of the various pieces of syntax. To begin we must consider some general concepts related to processing and meaning.

6.1 Processing Modes

As described in section 3, prescriptions may be processed in two ways. Each way is described as a *processing mode*. The modes are:

Verify	Examination of described entities only
Repair	Modification of described entities as required

The operational distinction between the two modes concerns the possibility of modification of the distributed system. In the first case, managed entities are examined, but no changes are made. In the second case, changes are made *if necessary*.

Unnecessary changes are never made. Suppose that a prescription describes a configuration in which a particular file exists with certain permissions. If the file already exists at a time when the prescription is processed in repair mode, the only possible change is a modification to the permissions of the file.

Processing mode is normally inherited across prescription activations. If prescription A is activated in Repair mode, and contains an activation of a prescription B, that prescription will also be activated in repair mode. Explicit narrowing may be used to override this default behaviour, as explained below.

6.2 Statement Forms

Most statements have only one form. The **foreach**, **disallow** and **require** statements, however, both have a general form and a more specific form with additional bits of syntax. The optional syntax for the more specific form provides distinguished information to identify the particular object(s) which are the subject of the statement. For all these statements, the additional information can be used to make verification more efficient. In the case of **require**, however, the additional information is necessary to resolve some ambiguities and permit automated repair.

For example, consider the following simple use of **require**:

```
require F: File in m.files {
  (
    M.mode = 0755
  )
}
```

The statement says that there must be a file in the collection with the mode `0755`. If there is no such file, automated repair is impossible. The problem is that there is no general way to determine whether the difficulty is that an object is missing from the collection, and should be added, or that some object in the collection has the wrong state and should be modified. The repairable form includes information which unambiguously identifies the object that the statement is about, so it is possible to determine whether the object is missing or mis-configured when there is a problem.

The extra form for **require** meets the objective of providing a syntactic distinction between statements which are suitable for automated repair, and those which are not.

There are also two forms of logical statement. The repairable form contains only relations in which the left side of the operator consists of a single data value. The non-repairable form may contain relations with general expressions on both sides of the operator (eg. `(A+B)=(C+D)`). In the repairable form, each relational expression implicitly identifies a particular data item (the one to the left of the operator) as the subject of the relation. The identified data item is the one that will be modified if necessary to effect repair.

All statements in the language *except* **require** and logicals provide enough information in all forms to enable repair.

6.3 Narrowing

One of the objectives for the language states that there must be a way to express intentions about automated repair syntactically. Accordingly, the language provides syntax for limiting processing to verification. Such a limitation is called *narrowing*.

Narrowing inhibits automated repair. Preventing automatic repair is desirable when a repairable form cannot be written (see the description of forms above) or when the purpose of a piece of description is to express constraints or preconditions. For example, imagine a precondition that says that a filesystem may not be imported if it contains files owned by `root`, with the setuid bit set in the mode. An automated repair process could remove offending files, but it is likely that the system administrators would prefer to have the problem reported as an error so human investigation could take place.

The language provides two slightly different ways of specifying explicit narrowing. Any statement may be narrowed by enclosing it in square brackets. Alternatively, any prescription may be declared narrowed by including the **narrow** keyword in the prescription definition. In this latter case, all activations of the prescription are narrowed, without the need to enclose them in square brackets.

Narrowing affects the processing performed on statements and prescriptions, but not their declarative meaning. When narrowed statements or prescriptions are processed, the mode is temporarily changed to Verify if it was Repair. Normal inheritance of mode across prescription activations still applies, so prescription activations in a narrowed block are processed in Verify mode. When explicitly narrowed statements are processed in Verify mode, the narrowing has no effect.

Implicit narrowing is performed on statements which are in a non-repairable form. A warning should be produced in such cases. The situation is similar to implicit type coercion in a traditional programming language.

6.4 Truth Value and Execution

The language has a combined declarative/procedural nature. In a declarative sense, a prescription is a straightforward description of a configuration state. In procedural terms, there are processing algorithms for prescriptions which handle both verification and repair.

The meaning of prescriptions must be given in both declarative and procedural terms. The declarative meaning is based on *truth values*, while the procedural meaning is given by an *execution* algorithm.

6.4.1 Truth Values

Every prescription and statement has a value of TRUE or FALSE at any point in time for any mapping of identifiers to actual items. The value is TRUE if and only if the described configuration state holds for the objects referenced by the identifiers. The derivation of truth value for every type of statement in the language will be described shortly.

Determining the truth value of a statement at a point in time requires that objects referenced by identifiers be examined. The means of examining objects varies with the types of the objects. Truth value determination is the basic purpose of execution.

The processing modes described earlier do not fundamentally affect the declarative meaning of any statement. They do affect the execution, however, and may thus influence the truth value derived.

6.4.2 Execution

Execution of a prescription or statement is the process of computing a truth value, possibly including modification of referenced objects in order to obtain the value TRUE. The execution algorithm for each type of statement will be described shortly.

Processing mode significantly influences execution. In Verify mode, the purpose of execution is to determine the truth value without any side effects. In Repair mode, the purpose of execution is to make changes to objects so as to obtain the value TRUE at the point of completion of the execution, if at all possible. In the case of verification, the execution itself does not change the truth value, but only computes it. In the case of repair, the truth value at the end may be different from the truth value at the beginning due to side effects of the execution.

Either truth value may be computed in either mode. In Verify mode, of course, the truth value computed is strictly dependent on the state of the managed objects. In Repair mode, the value TRUE will be computed if any automated repair steps can achieve that result, but the value FALSE will be computed otherwise. Repair failure may be due to inhibitions caused by narrowing, or other problems or errors.

A few general rules capture the main features of the execution algorithms:

1. Statements (and parts of statements, where appropriate) are processed in order of occurrence.
2. Only the processing required to compute a truth value is performed. This rule means that not all statements are processed in all cases.
3. Repair is atomic. See the next section for a description of this point.

6.4.3 Atomicity and Side Effects

Execution in Repair mode can have side effects intended to produce the value TRUE. If all goes well and the repair is successful, there is no problem. If the repair is unsuccessful, however, and a value of FALSE is computed, there is a question about outstanding side effects. The problem is that the execution may have failed to complete a repair step *after* successfully completing earlier repair steps.

To leave the managed objects in a partially repaired state is undesirable. For this reason, execution should be atomic with respect to side effects. When execution returns the value FALSE, all objects will be in the same state as before execution. When execution returns the value TRUE, the described configuration state will hold [3].

In a distributed, multi-processing environment, there are some potential problems with this atomicity policy. It may not be possible or appropriate to rollback all changes. During execution, other processes may make changes to objects after those subject to rollback. There may also be a large number of modifications to be reversed. Some modifications may not be reversible. Atomicity is a reasonable goal, but more work is needed.

7 Statement Descriptions

Each type of statement is described here in terms of truth value derivation and execution.

[3]This is from the perspective of the execution process. Other processes may interfere and cause these statements to be false.

7.1 Blocks

The meaning and processing of a block is the same regardless of whether it appears as an independent statement, or as part of another statement.

There are two types of blocks, distinguished by the way a truth value is derived from the truth values of contained statements. The types are syntactically differentiated as described earlier.

7.1.1 Truth Value

The truth value of a block is derived from the truth values of the contained statement(s). An empty block has the value TRUE.

An AND-block has the value TRUE iff every contained statement has the value TRUE.

An OR-block has the value TRUE iff there is one contained statement with the value TRUE.

7.1.2 Execution

In Verify mode, statements in the block are processed in order until the truth value of the block has been determined. In an AND-block, processing stops with the first statement to have the value FALSE. In an OR-block, processing stops with the first statement to have the value TRUE.

In Repair mode processing of an AND-block, each statement is processed in order. If the statement is narrowed, it is processed in Verify mode, and execution of the block stops if the statement has the value FALSE. Statements which are not narrowed are processed in Repair mode, so they will only have the value FALSE if all possible automated repair steps were unsuccessful.

Repair mode processing of an OR-block is more complicated. First the block is executed in Verify mode, to determine whether any repair is required. If the block has the value FALSE, then a second pass is performed, executing statements in Repair mode in order. As soon as one has the value TRUE (implying successful repair), execution of the block ends. If no statement can be successfully repaired, the block has the value FALSE.

7.2 Prescription Activation

A prescription activation serves the same purpose as procedure call in traditional programming languages. For processing, formal parameter identifiers are bound to the supplied actual arguments, and the block given in the definition of the activated prescription is processed with that mapping.

7.2.1 Truth Value

The truth value of a prescription activation is the truth value of the body block with actual arguments substituted for formal parameters.

7.2.2 Execution

A prescription activation is executed by creating the binding of formal parameters to actual arguments and executing the body block. Processing mode is inherited across prescription activation unless the activation is explicitly narrowed with square brackets, or the prescription is defined as narrowed.

7.3 Foreach

The **foreach** statement is a description of the state of a set of objects from the collection specified in the **in** part. The optional **with** clause allows a subset of objects to be selected from the collection by attribute value(s).

An identifier declaration is part of every **foreach** statement. The declared type must match the type of the objects contained in the collection. The scope of the identifier is limited to the block that acts as the body of the statement. For processing, the identifier is successively bound to each object from the collection which meets the selection criterion (an object meets the selection criterion if the **with** logical evaluates to TRUE for that object).

7.3.1 Truth Value

The truth value of a **foreach** statement is the logical AND of the truth values of the contained *block* with the declared identifier bound to the various acceptable members of the collection. For the statement to have the value TRUE, the block must have the value TRUE *for each* selected object.

If the collection is empty, the statement has the value TRUE.

7.3.2 Execution

A **foreach** statement is executed by successively binding the declared identifier to members of the collection, then executing the block. When a **with** clause is part of the statement, the supplied logical is evaluated for each member of the collection prior to binding. The object is skipped if the logical evaluates to FALSE.

In Verify mode, the block is executed with each selected object in turn until the collection is exhausted or the block has the value FALSE for some object.

In Repair mode, the block is executed for each object in Repair mode. It will only have the value FALSE if repair is unsuccessful. Should that be the case for an object, execution of the statement terminates.

7.4 Require

The **require** statement describes the state of at least one object which must exist in the collection specified in the **in** part. The optional object reference uniquely identifies the object which is described by the statement and will be subject to any repair actions taken.

An identifier is declared in the statement, as in the **foreach** statement. The identifier may be bound to various objects from the collection. The scope of the identifier is limited to the block contained in the statement.

7.4.1 Truth Value

The truth value of a **require** statement is TRUE if there is at least one member of the collection which can be bound to the identifier to make the contained block have the value TRUE. If the optional object reference is supplied, the referenced object must exist, and the block must have the value TRUE with the identifier bound to that object. If the collection is empty and the block is non-empty, then the statement has the value FALSE.

7.4.2 Execution

The form of the statement (with or without an object reference) is significant for execution. Without an object reference, repair is not possible and the statement is implicitly narrowed. With an object reference, repair is enabled, and shortcuts for verification can be taken.

First, consider the case of **require** without an object reference. Execution proceeds as with the **foreach** statement, except that the process can stop earlier. When an object from the collection is bound to the identifier, and the block with that binding has the value TRUE, then the statement has the value TRUE and execution stops. Due to narrowing, the block is always executed in Verify mode, and no repair is attempted.

Execution with an object reference is a bit more complicated. The first step is always a search through the collection for the object identified in the reference. The search algorithm varies depending on the nature of the collection. In some cases, access to objects by key may be directly supported. Regardless of the nature of the search, the first step terminates when an appropriate object has been found, or it has been determined that no such object exists in the collection.

In Verify mode, the absence of the referenced object implies that the statement has the value FALSE, so execution terminates when the object cannot be found.

In Repair mode, the absence of the referenced object implies that it should be created. The automated repair action involves the following steps:

1. Assign values to attributes as required by the object reference in the statement.
2. Assign default values to mandatory attributes which do not have values after the first step.
3. Create a new object with attribute values as determined in the previous steps, and insert that object into the collection [4].
4. Process the block of the **require** statement in Repair mode, with the identifier bound to the newly created object.

If the search for the referenced object is successful, then the identifier is

[4] Creation and insertion may be performed as a single unified action, depending upon the characteristics of the collection and the objects it contains

bound to that object, and the block is processed without change of mode. In Verify mode, the net effect is very similar to processing **require** without an object reference, but with the attribute requirements from the reference specified as logicals in the block instead. The only difference is a *potential* efficiency gain in those cases when a search can avoid examination of many of the objects in a collection.

For automated repair, the object reference provides a critical piece of information by identifying *which* object should be modified or created to effect repair. This information makes it easy to distinguish between a problem caused by some object having the wrong state, and a problem caused by the *absence* of a required object.

7.5 Disallow

The **disallow** statement describes part of a configuration in a negative way. It is a direct opposite to the **foreach** statement. It is also opposite to the **require** statement in the sense that the repair of **disallow** involves object destruction, while the repair of **require** may involve object creation.

The **disallow** statement implies removal and destruction as the appropriate repair action for offending objects. If destruction is not appropriate, the only option is to narrow the statement, then deal with the problem objects manually.[5]

7.5.1 Truth Value

A **disallow** statement has the value TRUE iff there is no object in the specified collection for which the contained block has the value TRUE.

7.5.2 Execution

Execution proceeds as with the **foreach** statement. In Verify mode, execution is the same as with **foreach**, with the sense of the block evaluation reversed. Thus the processing stops as soon as an object is found for which the block has the value TRUE.

[5] This destruction is equivalent to UNIX unlink, rather than active destruction, in cases where the distinction is significant.

In Repair mode, *every* object in the collection (satisfying the **with** clause, if present) is bound to the the identifier in turn. Any object for which the block has the value TRUE is *destroyed*.

7.6 If

The **if** statement functions like `if` in traditional programming languages such as C. The value of the contained *logical* controls the evaluation of the entire statement.

7.6.1 Truth Value

When the controlling logical expression evaluates to TRUE, the truth value of the statement is the truth value of the first block. When the controlling expression is FALSE, the truth value of the second block is the truth value of the statement. If the expression is FALSE and the optional **else** block is omitted, the statement has the value TRUE.

7.6.2 Execution

Execution begins with evaluation of the controlling expression, which is never subject to repair. Based on the obtained value, execution continues either with the first or second block. Processing is the same in either mode.

7.7 Logical

A logical expression is a legal statement describing the state of objects by describing the legal values for their attributes. In object-oriented fashion, the perceptible state of any component is assumed to be entirely represented by the values of attributes.

Logicals consist of combinations of relations with the standard logical connectives (AND, OR, NOT). Thus the processing of a logical is similar to the processing of a block. In the description given here, the details of logical and relational expression evaluation are not presented.

The objects described in a logical statement are referenced by identifiers defined outside the statement. These statements always occur inside a block, and a block is never processed unless all identifiers are bound to objects.

7.7.1 Truth Value

The truth value of the statement is just the truth value of the expression, when evaluated in a standard way.

7.7.2 Execution

The first execution step is always the evaluation of the expression. In Verify mode, no other action is required. In Repair mode, however, some modification of objects is required if the evaluation produces a value of FALSE.

The form of the statement is significant. If the logical contains non-repairable relations[6], then the logical is itself non-repairable and is implicitly narrowed.

The generality of boolean expressions makes automated repair tricky. The repair process has two phases:

1. Compute a list of attribute value changes which would cause the expression to have the value TRUE. The modifications on the list must be non-conflicting.
2. Apply the changes on the list produced by the first phase.

As an example, consider the following abstract situation. The logical is a simple conjunction of four relational expressions relating integer attributes to integer constants. It is expressed syntactically as:

```
( A.a = 1, B.b > 2, C.c < 14, D.d != 5 )
```

Now suppose that the attributes have the following values for a particular binding of objects to identifiers:

```
A.a = 27
B.b = 0
C.c = 85
D.d = 5
```

[6] Non-repairable relations are those which do not have a simple value on the left side of the relational operator. For example, `(A+B)=(C+D)` is non-repairable.

It should be obvious that the logical has the value FALSE. This example illustrates some of the difficulties of the first phase of automated repair. Note however, that one area of difficulty is avoided, because the expression is just a conjunction, so each relation must be made true.

The first relation is an example of the easy case. The important property of the relation is that it is *definite*. A single, specific value for the attribute is clearly stated. The change list begins with $\{ A.a \leftarrow 1 \}$.

The remaining relations are *indefinite*. With integers, it is easy to imagine that the repair algorithm can simply select an appropriate value. Thus the change list might become $\{ A.a \leftarrow 1,\ B.b \leftarrow 3,\ C.c \leftarrow 13,\ D.d \leftarrow 6 \}$. These selections may not be "best" on any reasonable scale of goodness, but will work. With strings, on the other hand, automatic selection of workable values becomes more complicated.

While simple conjunctions may turn out to be the most common type of logical in practice, the repair algorithm must be capable of dealing with much more complex expressions. When logical OR is used, the question of *which* relations to repair may arise. As with OR-blocks, order is significant. Thus repair is attempted on relations in the order in which they occur. Negation in logicals does not pose serious problems for repair.

There is no backtracking in the repair algorithms. They may fail to find a solution when one exists, but are never exponential in complexity. Consider the following case, for instance:

```
( B.b < 2 * C.c, C.c < D.d - 5 )
```

If the attributes have the following values, there is clearly an assignment that could be made to effect repair:

```
B.b = 25
C.c = 10
D.d = 10
```

The simple algorithm without backtracking will not find a solution in this case. It will first assign an acceptable value to `B.b` (say 19), then assign an acceptable value to `C.c` (say 4). The second assignment invalidates the first, but the algorithm does *not* make a second selection. Instead, the repair attempt would be judged a failure on final evaluation of the logical.

More powerful solution-finding algorithms could be applied to specifications written in the language. For configuration repair, however, this does not seem to be worthwhile. The simple algorithm will handle most cases in practice, is easily understood, and is guaranteed not to be exponential.

Once the first step of the repair procedure is complete, there may still be problems which prevent successful repair. For one, the attributes which must be modified may be *dependent* attributes. A dependent attribute is one whose value cannot be changed independently of other attributes or features of the object. For example, `size` is a dependent attribute of a file; its value is a function of the contents of the file.

Problems with dependent attributes may at least be identified through static analysis of logical statements. Other problems may be encountered only at the time when a repair operation is attempted. Some of these may be due to the multi-process nature of the system. In the absence of locking, some process other than the configuration manager process may have removed an object that the configuration manager attempts to modify.

Handling logical statements is at the heart of the problem of automated repair. There remain open questions about how it can be effectively done.

7.8 Narrow

A narrowed statement is simply a statement that has been distinguished for verification processing only. The truth value is not affected by the narrowing. In a sense, the statement is specified by the author as purely declarative in function.

Since repair is inhibited for narrowed statements, execution proceeds in Verify mode.

8 Future Work

The language design presented in this report is really only the point of departure for an exploration of practical configuration management in distributed systems. A great deal of work remains to be done. This section summarizes some of the major open questions and problems.

One of the most obvious limitations of the work presented here is the lack of validation through implementation and trial with real problems. The set of examples tried up to this point is too small to justify confidence that the language is adequate. In addition, a real implementation is needed so that unpredictable problems can be uncovered. Ongoing work is addressing these problems.

The issue of object definition and modeling has been largely ignored in this report. Some definition language is required. Also, an object model must be produced for any conventional system to which the language is to be applied. Producing such a model will involve a lot of very tricky decisions. For example, if the target environment is UNIX, we need to decide how i-nodes should be represented relative to files and directory entries. There are also questions about the attributes that should be exposed. A UNIX file clearly has a size attribute. It is also reasonable to talk about the checksum of a file, although the operating system does not maintain such an attribute.

One feature that will be needed in the definition language is a way to identify properties of attributes that are important for repair. For example, the size of a file is a dependent attribute, as noted earlier. That fact needs to be part of the specification of the file object, in order that static analysis of prescriptions can identify logical statements that will pose a problem for repair. Identification of candidate keys for objects in collections is another issue.

Related to the problem of object definition is the question of typing. In this report, typing has not been explicitly addressed. In fact, the examples rely on some tricky interactions between statements and typing through the inheritance hierarchy. Formal typing rules need to be devised.

The context in which prescriptions exist has been left outside the scope of this report, although some possibilities are suggested in the examples. Prescriptions are the analogs of procedures in traditional imperative languages. There needs to be some structure to enclose prescriptions, analogous to a program in a traditional language.

The problem of managing collections of software with support for mobility needs to be further explored. It is entirely possible that changes or extensions to the language would be required to adequately support that application. For example, some use of revision numbers as in doit [Fle92] may be necessary.

As noted earlier, the desirable goal of atomicity poses some problems which require further investigation.

A Grammar

Here is a BNF grammar for the language described in this document. Non-terminals appear in *italics*, specific terminal strings appear in **bold face**, and general terminals appear as regular text. Comments are not represented in this grammar, nor are all lexical elements. The grammar here is intended for human interpretation rather than automatic parser generation.

prescription ⇒ **prescription** id *param-list block*
| **prescription narrow** id *param-list block*
param-list ⇒ (*decl-list*)
| ( )
block ⇒ { *statements* }
| {| *statements* |}
statements ⇒ *statements statement*
| *statement*
statement ⇒ *block*
| *logical*
| *activation*
| *regular*
| *narrow*
logical ⇒ (*rel-exp*)
activation ⇒ id *arg-list*
arg-list ⇒ (*expr-list*)
regular ⇒ *require*
| *disallow*
| *foreach*
| *if*
foreach ⇒ **foreach** *decl* **in** id *block*
| **foreach** *decl* **in** id **with** *logical block*
require ⇒ **require** *decl* **in** id *block*
| **require** *decl ref* **in** id *block*
disallow ⇒ **disallow** *decl* **in** id *block*
| **disallow** *decl* **in** id **with** *logical block*
if ⇒ **if** *logical block*
| **if** *logical block* **else** *block*
narrow ⇒ [*statement*]
ref ⇒ < *key-exp* >

decl-list $\Rightarrow$ *decl* , *decl-list*
$\mid$ *multi-decl* , *decl-list*
$\mid$ *decl*
$\mid$ *muti-decl*
decl $\Rightarrow$ id : id
multi-decl $\Rightarrow$ *id-list* : id
id-list $\Rightarrow$ id , *id-list*
$\mid$ id
expr-list $\Rightarrow$ *expr* , *expr-list*
$\mid$ *expr*
key-expr $\Rightarrow$ *key-val* , *key-expr*
$\mid$ *key-val*
key-val $\Rightarrow$ id = *expr*
$\mid$ *expr*
rel-exp $\Rightarrow$ *relation binary-op rel-exp*
$\mid$ *unary-op rel-exp*
$\mid$ (*rel-exp*)
$\mid$ *relation*
relation $\Rightarrow$ *simple-value rel-op expr*
$\mid$ *expr rel-op expr expr* $\Rightarrow$ *sub-expr comp-op sub-expr*
sub-expr $\Rightarrow$ *sub-expr comp-op sub-expr*
$\mid$ value
$\mid$ (*sub-expr*)
value $\Rightarrow$ (*value*)
$\mid$ *simple-value*
$\mid$ *global-value*
$\mid$ *special-value*
$\mid$ *ref*
$\mid$ literal
$\mid$ **size** (simple-value)
simple-value $\Rightarrow$ id . *simple-value*
$\mid$ id
global-value $\Rightarrow$ **$** *simple-value*
special-value $\Rightarrow$ **#NIL**
$\mid$ **#ANY**

B Examples

This appendix carries on from the examples of section 4.1.

B.1 Printer Configuration

For a slightly different example, consider printer configuration at Hedgehog. Every printing service has a server machine on the network, and is accessed through the server. There are three categories of printing service in the company.

1. Company wide
2. Department wide
3. Local

Company wide services are available from every workstation. Departmental printers are only available from workstations assigned to the department that operates the printer. Local printers are available to workstations that are physically situated in a certain area, regardless of department. Also, any workstation may be granted access to any printing service for a special purpose.

Suppose that the following tables are defined:

Printer Table			
name **key** String	**alternates** LIST of String	**host** *Machine*	**device** String
color-laser	cl, full-color, lwc	huey.hh.com	/dev/lp1
adminhv	admin-high-vol	red.hh.com	/dev/lp
robin		enterprise.hh.com	/dev/robin
jay	lw310, lw3fl	tulip.hh.com	/dev/lw310
lwsales	sales-shared	rose.hh.com	/dev/ptr/sales

The **Printer** table contains data describing each individual printing service.

Assigned-Printers Table	
printer *Printer*	**department** *Netgroup*
color-laser	#NIL
adminhv	administration
lwsales	sales

The **Assigned-Printers** table describes the assignment of printers to groups of machines. A value of `#NIL` in the **department** column indicates that the printer should be assigned to *all* machines.

Printer-Access Table	
printer **key** *Printer*	**clients** LIST of *Machine*
adminhv	gilbert.hh.com, sullivan.hh.com, huey.hh.com
robin	enterprise.hh.com
jay	tulip.hh.com, daffodil.hh.com, magellan.hh.com
lwsales	gilbert.hh.com, tulip.hh.com

The **Printer-Access** table describes the assignment of printers to individual machines. This table is used to take care of local printers and special access requirements.

The following prescriptions describe printer configurations based on data in the tables above. The table describes the assumed binding of global identifiers.

Identifier	Binding
`$machines`	LIST of records in **Machines** table
`$printers`	LIST of records in **Printer** table
`$assigned-printers`	LIST of records in **Assigned-Printers** table
`$printer-access`	LIST of records in **Printer-Access** table

```
prescription printers()
{

  -- Every print service is exported by some host
  foreach P: Printer in $printers {
```

```
    foreach M: Machine in $machines
    with (M.name = P.host) {
        ExportsPrinter(m, P)
    }
  }

  foreach A: Assigned-Printer in $assigned-printers {
    if (A.department != #NIL) {
      foreach M: Machine in A.department.members {
        PrinterClient(M, A.printer)
      }
    } else {
      foreach M: Machine in $machines {
        PrinterClient(M, A.printer)
      }
    }
  }

  foreach E: Printer-Access in $printer-access {
    foreach M: Machine in E.clients
        PrinterClient(M, E.printer)
    }
  }
}

prescription ExportsPrinter(m: Machine, p: printer)
{
  require P: PrinterEntry <p.name> in m.printers {
         (
            P.alternates = p.alternates,
            P.mx = 0
            P.lp = p.device
            P.sd = "/var/spool/printers/"+p.name
        )
  }
}

prescription PrinterClient(m: Machine, p: printer)
{
```

```
  if (m != p.host) {
     require P: PrinterEntry <p.name> in m.printers {
         (
            P.alternates = p.alternates,
            P.rm = p.host.name,
            P.rp = p.name,
            P.mx = p.maxSize,
            P.sd = "/usr/spool/"+p.name
        )
     }
  }
}
```

B.2 Integrity Constraints

As noted in section 4.1, there are various consistency constraints which could be applied with some of the example prescriptions. In this section, a number of constraints are expressed in the language.

With consistency constraints, it is often the case that only *verification* processing is desired. When a problem is detected, it should be reported to a human administrator who can take action. Thus narrowing is used extensively in the following examples.

When a machine is to mount a filesystem via NFS (as described by the `mountsNFS` prescription, for example) the machine should be functioning as an NFS client. To check that a machine is operating as an NFS client, one can check for the presence of certain processes. Here is a prescription that describes the client configuration:

```
prescription narrow nfsclient(m: Machine)
{
  require P: Process in m.processes {
        ( P.name = "rpc.statd" )
  }
  require P: Process in m.processes {
        ( P.name = "rpc.lockd" )
  }
  require P: Process in m.processes {
```

```
        ( P.name = "ypbind" )
  }
  require P: Process in m.processes {
        ( P.name = "(biod)" )
  }
}
```

Another possible constraint is that imported filesystems should not contain files with the setuid mode bit set. The following prescriptions express that constraint for a filesystem:

```
prescription narrow setuid-safe(f: Filesystem)
{
  foreach D: Directory in f.server.files {
    foreach N: File in D.contents {
      ( N.mode.setuid = 0, N.mode.setgid = 0 )
    }
  }
}
```

It is important to note that the body block in the `setuid-safe` prescription is repairable (as can be determined by inspection). If the **narrow** keyword were removed, the prescription could be processed in Repair mode. In that case, the repair algorithm would attempt to unset any setuid or setgid bits set on files in the filesystem.

The above two constraints deal with managed items directly. It is also possible to express constraints involving only data, as the following prescription illustrates.

```
prescription narrow check-logical() {
  foreach L: Logical in $logicals {
    foreach F: Filesystem in L.parts {
      disallow Q: Logical in $logicals {
        (Q != L, Q.parts contains F)
      }
    }
  }
}
```

The constraint expressed by the above prescription is that a given physical filesystem should be part of *only one* logical filesystem. Note that if the prescription were processed in Repair mode (after removal of the **narrow** keyword), the only repair action that might be taken would be destruction of a `Logical`, ie. a record in the **Logical** table. Since that form of automated repair is unlikely to be appropriate, narrowing is used.

There are also some consistency constraints which apply to the problem of printer configuration. For example, we might want to ensure that the **device** attribute of a record in the **Printer** table always refers to a valid device special file. The following prescription expresses the constraint for a particular device d on a particular server s.

```
prescription narrow device-check(s: Machine, d: String)
{
  ( d ~= "^/*" )
  require C: CharSpecial <fullPath = d> in s.files {
    require D: DeviceDriver in s.installedDevices {
      ( D.name = "Printer", D.minor = C.minor,
        D.major = C.major )
    }
  }
}
```

The first logical statement in the above prescription uses the `~=`, borrowed from the Perl language. The operator compares a string against a regular expression.

It is almost certainly the case that there are reasonable constraints which cannot be expressed in the language.

References

[ACN92] Don Acton, Terry Coatta, and Gerald Neufeld. The Raven System. Technical Report TR-92-15, Department of Computer Science, UBC, September 1992.

[CN94] Terry Coatta and Gerald Neufeld. Distributed Configuration Management Using Composite Objects and Constraints. In *Second International Workshop on Configurable Distributed Systems*, Pittsburgh, 1994.

[CW92] Wallace Collyer and Walter Wong. Depot: A tool for Managing Software Environments. In *Proceedings of the Sixth System Administration Conference (LISA VI)*, page p. 153. USENIX Association, Berkeley, CA, October 1992.

[Fle92] Mark Fletcher. doit: A Network Software Management Tool. In *Proceedings of the Sixth System Administration Conference (LISA VI)*, page p.189. USENIX Association, Berkeley, CA, October 1992.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, NJ, 2nd edition, 1988.

[MP] Marc Majka and George Phillips. *tanis - system for boot-time machine configuration*. Department of Computer Science, UBC. online manual.

[RGL88] Mark A. Rosenstein, Daniel E. Geer, Jr., and Peter J. Levine. The Athena Service Management System. In *Usenix Conference Proceedings*, Winter 1988.

[RL91] Kenneth Rich and Scott Leadley. hobgoblin: A File and Directory Auditor. In *Proceedings of the Fifth Large Installation Systems Administration Conference*, page p. 199. USENIX Association, Berkeley, CA, September 1991.