Configuration Management Using Objects and Constraints

by

Terry Coatta
B.Sc. (Computer Science) University of British Columbia, 1985
M.Sc. (Computer Science) University of Toronto, 1987

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENT FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

in

THE FACULTY OF GRADUATE STUDIES
COMPUTER SCIENCE

We accept this thesis as conforming
to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA
April 1994

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

(Signature)

Department of Computer Science

The University of British Columbia
Vancouver, Canada

Date April 21, 1994

DE-6 (2/88)

Abstract

Distributed programming techniques have transformed applications into federations of cooperating semi-autonomous components. Complex interactions between these components create complex interdependencies which are quickly outstripping the capacity of human systems managers. Adding configuration management features to an application's components reduces the flexibility and portability of those components. This thesis develops a new model of configuration management designed to address these issues. To respond to the wide variety of situations in which configuration management is required, the model is universal, addressing with one formalism many different types of configuration management tasks. In order to reduce reliance on human systems managers, the model describes configuration management as an active process carried out by the computer itself. Finally, as the need for automated configuration management is most acute in distributed systems, the model provides for the distribution of configuration management activities.

This new model of configuration management is the basis for the design of a new configuration management tool, the Raven Configuration Management System (RCMS). RCMS provides an environment in which configuration management is handled orthogonally to the task of programming individual components. RCMS allows a programmer to group related components together and specify rules which govern their interactions. RCMS combines object-oriented structuring with declarative programming to produce a system which provides improved reliability and performance in the presence of an evolving environment. The core of RCMS is a new configuration programming language called the Raven Configuration Language (RCL).

This thesis describes the implementation of RCMS/RCL. This implementation demonstrates that the language constructs of RCL can be performed with acceptable efficiency, and it establishes the sort of services required to implement a universal, active, and distributed configuration management system. Finally, this thesis discusses several interesting implementation techniques that address particular problems which arose during the implementation of RCMS/RCL.

Table of Contents

List of Figures

Acknowledgment

Without supervisors there would be no theses: a maxim surely proved by this thesis. Gerald Neufeld has had the patience to let me explore and the strength to make sure that, in the end, the work was complete. Many thanks also to the rest of my thesis committee for the time and knowledge each has contributed.

To the guys from Room 360 (alas, it lives no more, save in our hearts), many thanks for encouragement and for the ability to introduce a note of levity into any situation. Special thanks to Don Acton for spending time helping me with debugging and revealing to me the inner secrets of the Raven run-time. Thanks, as well, to George Phillips for his many hours of hard work on WWW, so that when I grew tired of the work which I was supposed to do, I could experiment with something that was *really* interesting.

My parents, I am sure, had no idea what they were getting into when they set me off on this path by instilling in me a love of learning. There is a strong connection between the curiosity which drives research and the need to know what the Cat in the Hat is going to do next. They are, in addition I am sure, happy to see their 31-year-old son finally on his way to getting a *real* job.

My wife Louise is perhaps surprised more than anything else. Having endured a number of "next year" responses to the question "when are you going to be finished?", it may be difficult to accept that this "next year" is the last. She has managed to nurture, comfort, reassure, and support a hopelessly insecure graduate for a period of seven years. I only hope that I can return the gift.

CHAPTER 1

Introduction

Correct operation of any system consisting of multiple components depends on the well-established relationships between those components; wheels must be attached to axles, appliances must be provided with electricity of the correct voltage, and power must be supplied to IC's at the appropriate pins. In the broadest sense, a *configuration* is any collection of components and their relationships to one another. *Configuration management* consists of identifying the relationships which govern the behaviour of a system, recognizing systems in which those relationships do not produce the desired behaviour, and taking corrective action to remedy the problems. The engineer designing a new circuit, the programmer updating a `makefile` for a simple application, and the frustrated driver trying to avoid a collision in stop-and-go traffic are all engaged in configuration management. In the realm of computers, configuration management abounds. An operating system is a configuration of primitive functions, device drivers, and service providers. A large application program under development is a configuration of source files, object modules, and documentation. In a process-oriented operating system, an active

application program is a collection of processes, configured to communicate in a particular pattern. Each of these systems cannot function without configuration management.

In computer systems, the task of managing configurations is often performed piecemeal by programmers and systems managers. The programmer includes code which explicitly monitors changes in the system and responds to them. The systems manager manually ensures that an operating system has the correct device drivers installed or that system tables are updated to reflect the current system structure. Often the definition of acceptable configurations and knowledge concerning how to correct faults is stored principally *not* within computer systems, but within the humans who program and operate these systems. Furthermore, the acceptability of the system's behaviour is often monitored *not* by the computer, but by its operator. Even when configuration management tools exist, they are generally constructed in isolation and lack both a common interface and a common representation for configuration information. Such *ad hoc* configuration management techniques are growing increasingly cumbersome and error-prone as systems grow in size and complexity.

The lack of a comprehensive approach to configuration management leads to the development of systems lacking flexibility, possessing a multiplicity of configuration tools, and frequently relying on human intervention to compensate for changes in the computing environment. Wolfson et. al. [57] note that, in the absence of a configuration management system, the addition of further management functions to an application may involve "extensive applications programming." Kramer et. al. [30] emphasize that failure to separate configuration issues from applications programming results in systems which are "difficult to construct, debug, and maintain" and results in software components unsuitable for re-use.

Observation of any large computer system reveals that system managers spend a good deal of time carrying out routine tasks in response to machine crashes or network problems. And while

typical Unix systems do offer the programmer configuration management tools, such as `make` [18] and `SCCS` [1], there is little coordination between them, forcing the user to learn the idiosyncrasies of how each tool approaches configuration management. The proliferation of distributed systems has exacerbated these problems by introducing concerns of locality and consistency between sites. The solution to all of these problems is to create a general model for configuration management and develop a tool which embodies this model. This thesis undertakes that task.

1.1 Basic Concepts

A distributed system consists of software and hardware components. Each component has a state which, ideally, can be inspected and altered. The state of the component governs the manner in which it operates. For example, part of the state of a routing device is its routing tables. Altering the content of the routing tables can improve network performance, or it can result in a loss of network connectivity. Avoiding the latter requires that the routing tables of that particular device be consistent with both the routing tables of other such devices and the desired degree of network connectivity. Achieving the former requires monitoring network traffic and adapting to changes in the pattern of information flow.

Configuration information is most conveniently represented in terms of the states of the components. For example, the fact that a given module has been tested could be indicated by an attribute `tested` which takes on the values `True` and `False`. Relationships between components, in turn, are represented by references from one component to another. If file `foo.o` is produced from file `foo.c`, then an attribute `producedFrom` of `foo.o` and containing a reference to `foo.c` represents this relationship. The sorts of relationships that can be expressed are limited only by the interfaces offered by the components of a configuration.

Combining the ability to express relationships between components with the ability to express constraints over those relationships is the basis of configuration management. Consider the constraint, stated informally, that the date of last modification of `foo.o` must be greater than the date of last modification of `foo.c`. Anyone who is familiar with Unix will recognize that maintaining this constraint lies at the heart of the simple configuration management utility `make` [18].

1.2 Problem Definition

As noted in the previous section, the central issue in configuration management is maintaining relationships between the components of a system. Chapter 3 of this thesis examines a number of existing configuration management systems which address various aspects of the problem. This analysis reveals a significant flaw in the existing approaches to configuration management: no one has examined the problem of configuration management with any significant degree of abstraction. Existing configuration management systems represent particular solutions to particular problems. They are not derived from an underlying model of configuration management. Thus, the essential task in addressing the global issues of configuration management is to create a model of configuration management which is general, addressing in one formalism the wide variety of management tasks carried out by existing systems. Furthermore, this model of configuration management must be active in order to provide for automated configuration management. Finally, as the need for automated configuration management is most acute in distributed system, the model must be distributed. Chapter 2 develops this model.

Simply creating a model, however, is just the beginning. It must be shown that the model is viable. Thus, the model must be made concrete. In this thesis, this task has been sub-divided into two others. First, the model of configuration management from Chapter 2 is used to guide the definition of a new configuration management language. The abstract features of the model are

translated into specific language features. This new language is described intuitively in Chapter 4 and more formally in Appendix 1. Second, an implementation of the new language is provided in order to prove that the language constructs can be performed with acceptable efficiency. Sample applications developed in this language demonstrate the general nature of the system. Several sample applications are presented in Chapter 4. Chapter 5 and Chapter 6 discuss various aspects of the implementation of the language.

1.3 Boundaries of the Problem

The research whose results are presented in this thesis has as its goal the creation of a model of configuration management based on the concept of constrained relationships between the components of a system. The viability of the model is demonstrated by implementing a tool capable of expressing and maintaining these sorts of relationships and using that tool to implement several sample systems. In order to reasonably limit the scope of the research, the tool has been designed in the context of a homogeneous object oriented system called Raven [39], which is itself a research project at the University of British Columbia. The tool is referred to as the Raven Configuration Management System (RCMS). Despite the nature of the current implementation, neither the model nor RCMS itself are inherently restricted to this environment, and Section 6.6 of this thesis discusses using the tool in a more heterogeneous environment.

The concept of constrained relationships is very broad, and RCMS does not address all aspects of the issue equally. Research focussed on a number of features of RCMS. Providing a common mechanism for performing configuration tasks was highlighted because it provides an alternative to the diversity of *ad hoc* configuration tools. Constraint maintenance was considered important because it allows the automation of a number of routine system management tasks which are currently performed manually. In addition, it permits the construction of systems

which continuously adapt to changes in the computing environment, resulting in improved performance and reliability.

RCMS features a formal specification language in conjunction with the ability to actively monitor and repair the systems it manages. Correctness criteria for a system are expressed as constraints. A system is *valid* if all of its constraints are satisfied. RCMS actively monitors the state of all managed systems. When a system becomes invalid because one of its constraints is not satisfied, RCMS attempts to identify the source of the problem and take corrective action.

RCMS is not, however, suitable for systems with strong real-time requirements. Although RCMS was designed to be efficient and to provide reasonable interactive performance, it is not designed for reactive systems where delays between action and response significantly affect system behaviour. Such problems lie within the domain of control theory. And while RCMS can be used to improve the reliability of a system, it is not a substitute for fault tolerance provided through redundancy. Redundancy requires a tight coupling between active and backup components, which is beyond the scope of RCMS. Finally, there are some configuration tasks which can be expressed in terms of relationships and constraints, but which would exhibit poor performance using RCMS. For example, it is possible to specify version management in terms of the relationships between one version of an object and a subsequent version. However, without a high degree of optimization, possibly relying on the semantics of the objects themselves, RCMS could neither store nor access the various versions of an object with reasonable efficiency. The preponderance of other tools and systems [18][20][22][37][41][43][48][52] [54][55][57][58] that acknowledge these same sorts of limitations should dispel concerns that the problems addressed in this thesis are unworthy of research.

1.4 Thesis Statement

The principle thesis of this research is:

> *It is possible to define a model of configuration management which is general, active, and distributed. Furthermore, it is possible to realize a language/system based on this model with an efficient implementation.*

1.5 Thesis Contributions

There are three distinct areas in which this thesis makes significant research contributions:

(1) A new model of configuration management is defined. The emphasis in this model is on general, active, and distributed configuration management.

(2) A new configuration management language is introduced. The language, referred to as the RCMS configuration language, is based on this new model of configuration management.

(3) Implementing RCMS provides insight into the set of services which are required to support configuration management. The issues involved in providing these services with an acceptable level of performance are explored and several new solutions to these problems are presented.

The following sections of this chapter provide more detail about the nature of the research contributions made in each of these areas.

1.5.1 A new Model of Configuration Management

The RCMS model of configuration management is based on the observation that all managed entities can be viewed as objects and all management activities as the process of maintaining relationships between these objects. Those objects might be parts of a distributed application, cooperating to provide a particular service. They might be elements of a software development project: design specifications, source code, and documentation. Indeed, they could be any col-

lection of objects related to each other in any manner. However, by adopting a uniform point of view the RCMS model is able to address all of these types of configuration management tasks. Thus, the RCMS model is general.

The RCMS model also stresses that configuration management must be an active process. This requires that the configuration manager monitor the behaviour of the system in real time with the goal of detecting problems when they occur. In addition, the configuration manager must possess the ability to repair problems when they are detected. Only when these two requirements are met can the primary responsibility for configuration management shift from systems managers to the computers themselves.

Finally, the model provides a way to view configuration management in a truly distributed fashion. It introduces the concept of a *composite object*, an entity which tightly binds together the objects to be managed, the configuration management information, and the services related to managing those objects. These composite objects provide an ideal mechanism for distributing configuration management activities.

1.5.2 A New Configuration Language

The second major contribution of this thesis is a new configuration language whose design was guided by the RCMS model of configuration management. The significant features of this new configuration language are its object-oriented approach to configuration management, its flexible mechanism for identifying objects which are to be managed, and its constraint-based language for describing relationships between objects. In addition, the language is coupled with specific mechanisms for monitoring and repairing configurations.

1.5.3 A New Configuration Management System

The third and final area in which significant research contributions have been made is in providing an implementation of RCMS. First, this implementation establishes the viability of both the RCMS model and the RCMS configuration language. Second, the implementation establishes the sort of services required in order to implement a general, active, and distributed configuration management system. Finally, several interesting implementation techniques are used or proposed in order to address issues such as the efficient evaluation of quantified expressions, preservation of consistency through locking, improvement of the performance of component set computations, and improvement of the level of distribution of management activities.

1.6 Thesis Outline

Chapter 2 presents a basic model for configuration management and explores its requirements in detail. The chapter begins with a number of definitions for terms used throughout the thesis. The remainder of the chapter looks at the tasks a configuration management system is expected to perform and abstracts from this a set of requirements suitable for analyzing the capabilities of any CMS. A brief synopsis of how each of the requirements is addressed by RCMS is also provided.

Chapter 3 looks at existing configuration management tools and analyzes them with respect to this model. There is a wide variety of existing configuration management tools and this chapter selects a number of systems for detailed analysis with the goal of obtaining a representative sample of the sorts of tasks that configuration management systems are expected to perform. The chapter closes with a summary of the weaknesses of existing systems and further motivation for the development of RCMS.

Chapter 4 provides an overview of the RCMS configuration language. The chapter begins with a brief review of the Raven system on which RCMS is constructed. It then introduces the syntax and semantics of the language. The latter sections of the chapter present four sample applications which demonstrate the generality of the language and also introduce a number of specialized features not presented in the initial overview of the language.

Chapter 5 describes the current implementation: the RCMS compiler, run-time support routines, changes made to the Raven system, and new Raven classes added to support RCMS. All portions of the current implementation are described in detail, although particular attention is given to interesting problems which were encountered and the solutions for those problems.

Chapter 6 continues the discussion of the implementation of RCMS, but addresses a number of enhancements which have not yet been implemented. In particular, it looks at improving the performance of computing the components set of a composite object, improving the distribution of monitoring, providing a better repair mechanism, and applying RCMS to the problem of network management using the SNMP protocol.

Chapter 7 summarizes the results of this research. The major contributions of the research are presented and RCMS is analyzed with respect to the requirements which are detailed in Chapter 2. Weaknesses in the current implementation of RCMS are identified, and a section on future work outlines those areas in which further research would most significantly improve RCMS.

CHAPTER 2

A Model of Configuration Management

The central thesis of my research is that configuration management encompasses a broad range of activities and processes which can be expressed via a single formalism. In order to explore the ramifications of this proposition, this chapter defines a set of requirements which form a framework for the development of a new model of configuration management. These requirements also provide a means of analyzing the capabilities of any existing configuration management systems.

2.1 Definitions

In order to establish a context in which to present the requirements for a configuration management system, we need to begin with the definitions for several basic terms.

2.1.1 Objects

The need for configuration management arises only when there are a set of entities to be man-

aged. In this thesis, these entities are referred to as *objects*. It should not be assumed, however, that the use of the word "object" limits the applicability of the model to systems which advertise themselves as object-oriented. In fact, this definition of "object" is sufficiently broad that many things not normally considered objects, such as Unix™ files, satisfy the definition. An object, then, is simply an entity within a computer system that can be referred to, contains information, and is associated with a set of operations for accessing and modifying that information.

2.1.2 Invocation Syntax

In order to manage objects, the configuration system must be able to access and modify the contents of those objects. For the sake of uniformity, these actions are always expressed as the *invocation* of an operation on an object. An invocation is represented as:

```
<object identifier>.<operation name>(<parameters>)
```

For example, suppose the variable `X` contains an identifier for the file `sort.c` and the operation `lastModDate` returns the date of the last modification to a file. The value of the last modification date of `sort.c` is expressed:

```
X.lastModDate()
```

Actual systems typically possess many different syntactic mechanisms for expressing an invocation such as this and a wide variety of implementations for performing the actual task. However, a single syntax is convenient for discussion and, furthermore, suggests the underlying similarity of apparently dissimilar actions.

2.1.3 Location Transparency

In order to simplify the presentation of configuration information, it is assumed that object identifiers are location-transparent; that is, an object identifier may be passed from one location to another without the need for explicit translation. As with object invocation, actual systems have

a variety of ways of identifying a given object. Some of those representations are location-transparent and others not. For example, the *file descriptor* used within a Unix program to identify an open file cannot be arbitrarily passed from one process to another. However, assuming appropriately mounted file systems, the absolute path name for that file can be. The assumption of location transparency does not imply that this model of configuration management requires a system which provides location transparency as a basic feature. It simply separates concerns over location from configuration issues. Location transparency can always be provided on top of existing services. For example, protocols such as SNMP [6] and CMIP [25], which provide uniform access to management information in heterogeneous systems, could provide the underlying services that RCMS requires.

2.1.4 Effects of Failures

Because the model is intended to apply to distributed computer systems, the question of partial failures is significant. For example, it is well known that when one machine communicates with another, it is difficult to distinguish the failure of the communication system from the failure of the other machine. The model makes no specific assumptions with regard to the nature or effects of failures in the computer system. Interpretation of failures is the responsibility of the user of the configuration system. Section 4.4.4, for example, highlights how the interpretation of failures affects the design of systems in RCMS.

2.2 Configuration Management Requirements

Provided with the basic vocabulary of the previous section, we can now establish a means of exploring the different abilities of configuration management systems. Any configuration man-

agement system (CMS) can be assessed in terms of the following nine fundamental requirements:[1]

(1) Identification

(2) Control

(3) Specification

(4) Monitoring

(5) Repair

(6) Non-interference

(7) Integration

(8) Distribution

(9) Performance

The sections which follow describe each requirement in turn and the motivation for asserting that it is fundamental to the operation of a configuration management system. This descriptive material is followed by an brief explanation of how the requirement is handled by make, and finally, a synopsis of how the requirement is addressed by RCMS.

2.2.1 Identification

A *configuration* is a collection of objects which are related to one another and for which there are criteria to establish whether that collection of objects is functioning correctly. The most fundamental requirement of a CMS is, therefore, that it be able to identify the objects which it manages and identify the objects which are part of a given configuration. An object which is part of a configuration is called a *component.*

1. These features were identified while developing RCMS, but were influenced by the approach to management used by the ISO [24][46]. The features identified by the ISO as important to management services differ from these because ISO management services include matters such as accounting and auditing which are not part of configuration management.

Consider, for example, the archetypal configuration management tool in the Unix environment: `make` [18]. The configuration information which `make` uses is maintained as a text file known as a *makefile*. Figure 1 shows a simple makefile. The objects which `make` manages are files, and

```
CC = cc
CFLAGS = -g

SRC = iso.c chord.c grid.c
OBJ = grid.o iso.o chord.o

chord : $(OBJ)
   $(CC) $(CFLAGS) -o chord $(OBJ)

grid.o : grid.c
   $(CC) $(CFLAGS) -c grid.c

iso.o : iso.c
   $(CC) $(CFLAGS) -c iso.c

chord.o : chord.c
   $(CC) $(CFLAGS) -c chord.c
```

Figure 1 – Makefile for a Simple Program

they are identified by name. Explicit enumeration identifies the files of a particular configuration. A tool more flexible than `make` might allow the set of components to be specified in terms of some property of the objects, for example:

```
"all files in my directory which contain C source code"
```

In RCMS, the components of a configuration can be identified by either enumeration or by set membership functions expressed in a restricted version of first order predicate logic, or by a combination of the two. The fundamental elements of the expressions which define membership functions are object invocations, relational operators, the usual complement of logical operators, and universal/existential quantification. The composition of configurations defined in terms of such membership functions can vary in response to changes in the state of the system.

2.2.2 Control

A CMS must be able to create, modify, and destroy objects and configurations. Consider, once again, the makefile shown in Figure 1. The rule for each object file indicates how it can be produced from the corresponding source file. `Make` satisfies the control requirement because arbitrary UNIX programs can be executed from within the makefile. This ability to execute programs is a limited form of the operation invocation mechanism discussed in Section 2.1.

RCMS is integrated with the Raven system. The entities which it manipulates are Raven objects, and it has complete access to the features of the Raven system to carry out its tasks. The syntax for object invocation given in Section 2.1.2, is, in fact, precisely the syntax used in Raven, and hence, in RCMS. Because Raven is a uniformly object-oriented system, object invocation is the only access/modification mechanism required.

2.2.3 Specification

A CMS must be able to record specifications which identify the valid states of a configuration or which govern the interactions of its components. A specification need not be declarative. A program written in an imperative language, controlling some collection of objects, is as much a specification as is a set of logical constraints. Of course, the ease with which a specification is written, debugged, and maintained is a key factor in assessing the specification capabilities of a CMS.

The specification language used by `make`, for example, expresses dependencies between files and rules for updating files. In addition, `make` offers a number of features such as macro and rule definitions which allow makefiles to be written more succinctly.

In RCMS, each configuration possesses a specification which states the relationships between its components. These relationships are expressed as constraints which take the form of logical

expressions. For example, part of a specification emulating `make` would contain a constraint expression such as:

```
for all X X.dependsOn().lastModDate() > X.lastModeDate()
```

A configuration is said to be *valid* with respect to a given specification if all of the constraint expressions in the specification evaluate to *true*.

2.2.4 Monitoring

A CMS must monitor changes to components which can affect the state of a configuration. Affected configurations must be validated to ensure that specifications are not violated. Each specification is therefore associated with a *check set* of events for which the CMS monitors. When an event from the check set occurs, the CMS checks to see whether the configuration still satisfies its specification.

Initially, it may seem that the check set should simply consist of the occurrence of any operation invocation on a component which results in a change in the state of that component. Certainly, this results in the CMS being aware of any changes which might invalidate a configuration. However, a variety of motivations exist for using a more precise check set. For example, given a specification intended to emulate `make`, re-evaluation on every change to a source file is obviously undesirable. The check set for such a specification should be an event which indicates that the programmer is ready for the various components of the application to be re-built. Fault recovery and optimization may also lead to variations in the check set. Fault recovery requires events such as communications failures to be in the check set. Optimization may produce a check set which is tailored to the specification, to minimize re-evaluation of the specification.

Since `make` is only active when run by a programmer, its ability to monitor a configuration is limited. In effect, its check set consists solely of the event of being executed. Once active, `make`

engages in validation by identifying files which are out of date. A file is considered to be out of date if it depends on a file whose last modification date is greater than it own. Figure 1, for example, shows that `chord.o` depends on `chord.c`. If `chord.c` is edited, its last modification date will be updated, and the next time `make` is run it will notice that `chord.o` is out of date.

RCMS implements the check set by tagging each configuration with information about how monitoring is to be carried out. There are three basic types of monitoring in RCMS: watching for invocations on components, waiting for a communication error involving a component, or simply checking the state of the entire configuration periodically.

2.2.5 Repair

A CMS must either abort changes which would lead to the violation of a configuration specification or determine a series of actions which can restore the configuration to a valid state. These two modes of repair are referred to as *roll-back* and *roll-forward*, respectively. The former approach assumes support for recoverable transactions.[1] The CMS simply aborts transactions which result in invalid configurations. The latter approach presents a variety of possibilities. Given the complete semantics of the components and a suitable formal specification, the CMS could automatically compute the required repair actions. More realistically, however, the CMS requires information from the programmer providing some level of detail about the repair process.

Repair in `make`, for example, is roll-forward. The writer of a makefile is responsible for providing a set of Unix commands whose execution regenerates a target file from those files on which it depends. In Figure 1, for example, each object file is listed as depending on a corresponding C

1. Atomic transactions are not required, rather just the ability to roll back the states of modified objects to their previous values.

source file. Each dependency is followed by a line containing a command which runs the C compiler and regenerates the object file from the source file.

RCMS provides both roll-back and roll-forward repair.[1] Raven provides support for recoverability and RCMS is integrated into its recovery sub-system. Roll-forward repair in RCMS is very simple: the programmer is responsible for providing a set of repair scripts, written in an imperative language, which transform the configuration from a invalid state to a valid state.

2.2.6 Non-Interference

A CMS should not introduce inconsistencies into the system when carrying out management activities. The components of a configuration may have been designed with certain invariants assumed. CMS activities should not violate these invariants.

Suppose that object `A` has been designed to accept an input value, perform a computation on it, and then send it to another object. One of the assumptions made during the implementation of `A` may have been that the sending of the computed value always succeeds. Suppose further, that the CMS needs to bind a new component to the output of `A`. To do so, it removes the old binding, then establishes the new one. If the CMS continues to allow `A` to receive input values during this process, then it is possible that `A` will send an output value at a time when there is no object bound to the output of `A`. The design invariant has been violated.

As with many Unix tools, `make` does not really address the issue of non-interference. It is left to the user to ensure that his actions do not interfere with those of others. Because of this, a number of other tools have been developed whose purpose is to provide more structured coordination

1. An example of the use of roll-back repair appears in Section 4.4.1 and an example of roll-forward repair in Section 4.4.2.

between users. For example, both SCCS [1] and RCS [53] coordinate access to files which are being modified by a group of individuals.

RCMS addresses the non-interference issue by locking the components of a configuration during the validation/repair phase. This resolves some conflicts. However, creation/deletion of members of a configuration may require initialization/finalization actions. These additional consistency actions must be incorporated into repair scripts and may require additional features be added to the objects used in a particular configuration.

2.2.7 Integration

A CMS must minimize the changes to application software required to take advantage of configuration management services. The requirement for additional application-level code noted in the previous section causes a direct conflict between the non-interference requirement and the integration requirement. Ideally, application components require no modification to be used in the context of the CMS. Even if preservation of application consistency requires additional code, the CMS should still handle monitoring, operation invocation, specification evaluation, and repair activities. Use of CMS facilities, thus, requires only a small additional effort by the programmer.

Integration is only minimally addressed by `make`. The only management service it offers is maintaining date-based dependencies between files. Any management services required by an application must be expressed in terms of such dependencies if `make` is to prove useful. So, for example, maintaining a particular directory structure is a management task which cannot easily be handled by `make`.

RCMS is very tightly integrated with the underlying Raven system. Raven objects may be directly used within RCMS. Configurations defined in RCMS may be directly used within

Raven programs. The Raven run-time support system has been extended to provide a general purpose object monitoring mechanism. The Raven communication support system had been modified so that RCMS can detect communications failures. Even the RCMS language has been tailored so that there is a sense of uniformity between it and the Raven language.

2.2.8 Distribution

A CMS should minimize the difficulty of using configuration management services in a distributed environment. Given the continuing popularity of personal workstations and the ever-present need to share information, distributed systems have become increasingly commonplace. The complexity of distributed systems demands powerful and flexible configuration management services. Reliance on communications networks increases the need to adapt to changes in usage levels and to respond to communications failures. The degree to which a CMS meets this requirement can be assessed in terms of the difficulty of adapting it to a distributed system (if it has not been designed for distributed use), the effect on performance of increasing system size, and the ease with which it responds to partial failures of the distributed system.

Truly distributed versions of `make` do not exist. However, there do exist versions in which a single site computes the work which needs to be done to update a set of files and then farms that work out to a number of sites. Of course, in order to operate in such a manner, `make` relies on the presence of other distributed services such as the file system.

RCMS was designed with distributed systems in mind. Configuration information is not maintained in a central database. Rather, each configuration is directly associated with the constraints and repair actions which govern it. Distribution of the configurations thus results in distribution of configuration management.

2.2.9 Performance

A CMS must minimize performance degradation during normal operation of a system and provide satisfactory performance when carrying out management operations. In both cases, performance can be quantified, but it is difficult to establish a quantitative set of requirements because the fundamental issue is usability; if the system's performance is acceptable to its users, then the performance requirements are met. Of course, it is possible to measure the performance of individual configuration management systems so that different systems can be compared.

As `make` is only active when executed by a user, it has no impact on the performance of the system in normal conditions. When active, `make` has very good performance. This is a result of a reasonably long history during which a variety of optimizations and enhancements have been made. For example, the GNU version of `make` is capable of performing the updates for a given makefile in parallel.

An effort has been made to ensure that RCMS attains reasonable performance levels. Specifications are compiled rather than interpreted. Several performance bottlenecks have been detected and addressed. However, because RCMS actively monitors the elements of a system and has been designed with distribution in mind, it necessarily imposes more overhead than systems which do not provide these features. Only further experience with the system will determine whether it approaches the goal of usability.

2.3 A New Model of Configuration Management

In the preceding sections, RCMS is described in terms of the manner in which it addresses nine fundamental requirements for configuration management. In this section, that approach to configuration management is presented in a consolidated way as a new model of configuration management.

The basis of this model is a uniformly object-oriented view of computer systems. That is, the domain of the configuration management system is assumed to be a finite collection of objects which are denoted as the set $\{O_i\}$. Each of these objects have a state S_i, which is simply a vector of values. Each value is either an integer or a reference to another object. The state of an object is accessed via a set of attributes and operations. Access to an attribute of an object O is represented as: $O \cdot attributeName$. Reading the value of an attribute does not change the state of an object. Writing the value of an attribute changes one component of the state vector of that object. Invoking an operation on an object O is represented as: $O \cdot operationName(p_1, p2, ..., p_n)$. There are two varieties of operations, read-only operations and read/write operations. A read-only operation does not change the value of the state vector of an object which it is invoked on. Either variety of operation may optionally return a value to the invoker. A change in the state of an object O_i is said to change the observable state of an object O_j if it causes the value of an attribute or the value returned by an operation of O_j to change.

Within this system, configurations are represented as objects. That is, the set of objects $\{O_i\}$ is composed of two varieties of objects: elementary objects and configurations. Configurations are a special class of objects. In addition to attributes and operations, each configuration is associated with a boolean function $M_i(X)$ which describes the membership of the configuration. The set of components of a configuration O_i is defined as: $\{X|\ (X \in changeSet(O_i) \wedge M_i(X))\}$. The function $changeSet(O_i)$ returns the set of objects which can affect the observable state of object O_i. In order to define the change set we first need to define a simpler function $directSet(O_i)$. This is the set of objects which are directly referenced by an object O_i and is simply the set of objects for which there are references in the state vector S_i of O_i. The change set of an object O_i is then defined as:

$$changeSet(O_i) = \{X|\ (X \in directSet(O_i) \vee \exists Z, Z \in directSet(O_i) \wedge X \in changeSet(Z))\}$$

Each configuration O_i is also associated with a list of constraints $(C_{i1}, C_{i2}, ..., C_{in})$ which represent

the specification of the configuration. Each constraint is a boolean function expressed in first order predicate logic. The domain of quantification for these expressions is the change set of the configuration. Access to object attributes or operations are permitted within a constraint, but operations must be read-only. Assuming that the system is static (that is, that the states of all the objects in the system do not change), a configuration is valid if and only if all of its constraints evaluate to *true*.

Each configuration is also associated with a monitoring function. The specific functions used in RCMS are described in Chapter 4, but from the point of view of the model the monitoring function can simply be viewed as defining a sequence of times at which the configuration should be validated, that is, have its constraints evaluated relative to the current state of the system. The semantics of the model are that at a validation time for a configuration O_i, the objects of $changeSet(O_i)$ are locked. This means that the state of these objects cannot change. The current membership of the configuration is evaluated using the change set and the membership function $M_i(X)$. Those elements of the change set which are not members of the configuration are then unlocked. The constraints $(C_{i1}, C_{i2}, ..., C_{in})$ of the configuration are then evaluated in order. If any of the constraints evaluates to *false*, then the configuration is in an invalid state and must be repaired. Repair information is not represented in the model because that would require an axiomatization of the underlying repair language, which is not possible in RCMS currently. The cycle of validation and repair continues until the configuration reaches a valid state. At that point, the members of the configuration are unlocked. Thus, during the entire validation/repair phase, the states of the components of the configuration are unchanged except for repair activities.

2.4 Summary

This chapter began with a set of definitions which established the basic context for a discussion of configuration management at a relatively abstract level. A series of nine requirements for configuration management were then developed. These requirements provide a framework for exploring the capabilities of existing configuration management systems and provide a guide for the development of a new model of configuration management. Indeed, the RCMS model of configuration management is described by considering the manner in which each of these requirements is addressed by RCMS. Finally, at the end of the chapter, all of the elements of this new model of configuration management were brought together and presented as a whole in Section 2.3.

Chapter 3 takes the requirements which have been developed in this chapter and examines a number of other configuration management systems, revealing areas in which they do not address. The weaknesses in these other systems emphasize the strengths of the RCMS model: its general applicability, active management, and suitability for a distributed environment.

CHAPTER 3

Related Work

The need for configuration management originated with the creation of the first computers. Initially, the need could be met without much assistance from the computer itself. After all, there is little that the computer can do to help you keep your stacks of punched cards or paper tape reasonably organized. As computer systems became more powerful and more information was maintained on-line, however, there was a growing need for management applications, particularly with respect to software development. In time, this need grew to include a wide variety of tasks performed on the computer. This chapter examines how configuration management tools have developed to meet these needs.

3.1 First Generation Configuration Tools

An archetypal first generation configuration management tool is `make` [18]. `Make` assists the programmer in maintaining temporal consistency among files. The input to `make` is a structured

text file, usually called a *makefile*. A line of the form:

```
<filename1> : <filename2> <filename3>...
```

indicates that the file with name `filename1`, called the target, depends on files named `filename2`, `filename3`, etc. It is followed by a series of indented lines which indicate how to regenerate `filename1` using files `filename2`, `filename3`, etc. The purpose of `make` is to ensure that all files listed as targets are up-to-date. File `B` is out-of-date with respect to another file `A`, if `B` depends on `A` and the most recent modification to `A` occurred after the most recent modification to `B`. By examining modification dates, following dependencies, and recording update rules, `make` deduces a set of commands which causes all targets to become up-to-date.

Although make offers a variety of features[1] which enhance its usefulness, it is a conceptually simple tool. The objects which it manipulates are files. Only one type of constraint may be specified, namely that the last modification date of one file must be greater than that of another file. The monitoring strategy it adopts is the simplest possible: wait to be invoked by the programmer. Its repair strategy is straightforward: execute a series of commands supplied by the programmer. Because of its simplicity, most of the other requirements noted in Section 2.2 either do not apply or are trivially satisfied.

Increasing complexity in the software development process was the impetus driving the creation of a number of other configuration management tools which have followed in the wake of `make`. Tools such as the Configuration Management Assistant [41], IPSEN [55], and SySL [54] concentrate on providing a means to express structure and constraints on structure. SySL is a representative of this variety of application; let us look at it in more detail to see to what extent it satisfies the CMS requirements.

1. For example, it supports rules based on files suffixes, allowing a set of structurally similar dependency clauses to be replaced by a single rule.

SySL is a collection of related tools designed for the manipulation of structural information. The SySL language allows declarations of classes, structures, systems, and assertions. A structure is very much like a type definition. For example:

```
STRUCTURE viewer IS
   menu_setup,
   event_handler,
   operations,
   display,
   user_documentation
END STRUCTURE
```

defines a system composed of five components. The first four components are software modules. The fifth is documentation. An actual system satisfying these requirements is specified as follows:

```
SYSTEM sun_viewer : viewer IS
   menu_setup => suntools_setup,
   event_handler => suntools_event_handler,
   operations => sun_viewer_operations,
   display => sun_viewer_display,
   user_documentation => sun_viewer_doc
END SYSTEM
```

Groups of systems of the same type are explicitly grouped together with the class construct:

```
CLASS viewers IS (sun_viewer, apollo_viewer)
```

Constraints on systems are described by way of the assertion construct:

```
ASSERT viewer: up_to_date(user_documentation)
```

The type system of SySL is hierarchical and permits inheritance. This permits the decomposition of large systems into successively smaller and more manageable elements. The type system is extensible in that the fields of a structure are not constrained to a predefined set of possibilities. New types of relationships are created simply by adding new fields to structure declarations.

The system structure descriptions of SySL represent a database of configuration information. A syntax-driven editor permits modification of structure descriptions. A compiler translates the descriptions into a compact form suitable for manipulation by the computer. A viewer displays the information in a variety of formats. A system builder extracts dependency information from the description and produces a makefile. A verifier ensures that the constraints specified within the descriptions are satisfied.

The basic objects which SySL manages are files. The typically minimal semantics of files (read, write, seek, ownership, creation date, last modification date, etc.) limit the expression of relationships between them. Specialized programs which examine the contents of a file can provide more refined interfaces, but this results in the semantics of the system residing in the tools rather than in the configuration specification. Lack of uniformity in such tools results in diverse representations for configuration information. Like `make`, SySL provides no monitoring facilities. Invocation of the verifier is under programmer control. And while SySL can generate makefiles, it lacks any general purpose repair mechanism. The lack of monitoring or general repair facilities means that non-interference is not an issue and distribution reduces to the task of providing access to the configuration database. Integration is not well-handled due to the need for specialized tools to provide more extensive semantics for files.

3.2 Management Protocols

Software development has not been the sole force behind the development of configuration management tools. The proliferation of computer networks has created a host of concerns which also fall within the realm of configuration management. Often, networks use equipment from several different manufacturers and connect diverse computer systems. Consequently, configuration management in the context of networks was initially directed towards the establishment of standards for representing and exchanging configuration information. These standards serve as a platform for the construction of tools which transparently manage the diverse components of the network.

There are two significant network management protocols: The Simple Network Management Protocol (SNMP) [7] and the Common Management Information Protocol (CMIP) [25]. The former is used in TCP/IP-based internets and its design has tended to be driven by immediate application needs. The latter is an ISO standard whose design has been principally guided by a desire to establish a complete framework for network management. Despite the differences in the forces which have guided their development, the basic concepts of both are similar enough that it is sufficient to examine just one. The current popularity of SNMP seems reason enough to concentrate on it.

In fact, SNMP is just one of a trilogy of standards that together define a network management model for TCP/IP based networks [46]. The other elements of this trilogy are the Structure of Management Information (SMI) [44] and the Management Information Base (MIB) [45]. One of the principal concerns of SNMP is to provide a uniform network management environment despite the fact that networks are composed of various sorts of equipment, such as hosts, bridges, and routers, produced by a number of independent manufacturers. Furthermore, many different software components are responsible for the implementation of various services. If a

network management tool is to operate in such an environment, one of the first hurdles which must be overcome is standardization of the management interfaces for managed entities (referred to as *nodes* in SNMP). This is the purpose of the MIB and SMI standards.

Management requires access to information. One cannot ensure that a router is working correctly unless one can examine its statistics on messages received and transmitted and alter its routing tables. The first step in accessing information is identifying what information is relevant. A MIB definition fulfills this need by specifying globally unique names for each attribute of a managed node. In addition, a MIB definition organizes management information through the use of data structures such as lists and tables.

The SNMP MIB standard [45] is a MIB definition which exists by virtue of agreement between various national and international standards bodies. It defines a minimal set of attributes appropriate for management of TCP/IP-based networks. More extensive MIBs may be defined by other organizations, in particular, companies developing network devices and network management tools. In order to provide unique names for all attributes of managed nodes, such organizations must cooperate with standards organizations to maintain a consistent global namespace.

The SNMP MIB definition divides management information into several categories. The standard additionally defines, for each variety of node, the categories it must support. For example, the `system` category must be supported by all managed nodes. It contains variety of generic management attributes. An example member of this category is the attribute `sysDescr`. It is a text string describing the node. Its fully qualified name is 1.3.6.1.2.1.1.1. Armed with this globally unique name for the attribute `sysDescr`, and the further knowledge that all nodes conforming to the standard implement this attribute, one is now in the position to query such a node and obtain the value of its `sysDescr` attribute. Thus, we arrive at SNMP itself.

SNMP is a protocol describing how management information is actually accessed. At the application interface level, it is a very simple protocol. There are four commands supported:

(1) get: retrieves the value of a specific attribute or set of attributes

(2) get-next: retrieves the value of a specific attribute or set of attributes through traversal of the attributes in an order defined by the MIB.

(3) set: used to change the value of a specific attribute or set of attributes

(4) trap: sent by a node to a manager to indicate the occurrence of an unusual event

The actual protocol describes in detail how these requests should be transmitted to a managed node and how it should reply, but at the application level, a support library simply provides a set of routines corresponding to these commands. For example, the support library would provide a routine, which it might name `snmp_get()`, providing access to SNMP's get operation. Such a routine would take the fully qualified name for an attribute and a reference to the managed node, and return the value of that attribute for that node.

SNMP enjoys considerable support in the world of TCP/IP networking. Almost all networking products now support SNMP, and there are a number of companies producing management software which relies on the uniform management environment which SNMP provides. However, it should be clear that SNMP, and by analogy CMIP, address only the identification and control requirements of a complete configuration management system. Handling the remaining seven requirements is a task relegated to applications built on top of the services supplied by SNMP. Thus, RCMS and SNMP are focussed on different aspects of the configuration management problem. RCMS is primarily concerned with structuring and supporting configuration requirements through the automatic maintenance of constraints. SNMP operates at a more primitive level: Its sole purpose is to provide, in the context of a distributed and heterogeneous computing environment, uniform access to status information and a uniform control mechanism. In light of

this, SNMP and RCMS should be seen as complementary. Indeed, Section 6.6 provides a brief discussion of how RCMS could make use of the identification and control features of SNMP to create a management system which did not rely on the Raven system.

3.3 Advanced Configuration Management Tools

The past decade has seen a great deal more attention paid to configuration management. Focus has shifted from small-scale applications such as `make` to more comprehensive tools and environments, designed with configuration management concerns included from the outset. In this last section of our survey of related systems, we examine three systems developed during this period:

(1) Grapple [22], a software development tool based on planning software originally developed in the context of research into artificial intelligence.

(2) Conic [32], a language and execution environment designed specifically for the development of re-usable software components which can be easily configured into applications.

(3) Network Management System (NMS) [57], a network management application designed to automatically monitor and re-configure components of a network.

In Section 3.1, we noted that increasing complexity in the process of software development led to the creation of tools such as SySL [54], which addressed the problem by providing a means for expressing structure and relationships between the various components of a software development project. Software development is concerned with more than structure, however. Models of software development not only describe how objects should relate to one another, but also the processes by which such objects are created or derived from each other.

3.3.1 The Grapple System

Grapple steps beyond the capabilities of structural systems by attempting to codify the software process itself. Grapple borrows the planning paradigm from AI as the means for achieving this. In planning, there is a structural component, the *state schema*, which describes the objects that are manipulated and their relationships to one another, and a process component, the *operators*, which describe actions that modify the global state by creating, destroying, or modifying objects in the system. Operators are a powerful descriptive tool. Each operator is named and possesses a set of effects describing how that operator changes the state of the system. The *goal* of an operator identifies its primary effect. In addition, an operator may also possess sub-goals and preconditions. Sub-goals allow hierarchical decomposition by expressing higher-level operators in terms of simpler goals. Preconditions are constraints on the state of the system which must hold in order for application of the given operator to be legal. Given the state of the world and an initial goal, a planner seeks to derive a sequence of operators which result in the satisfaction of that goal.

In Grapple, an operator represents an element of a software development process. Huff and Lesser [22] describe a simple development process such as is frequently used for small software projects in a Unix environment. The most fundamental operator is `release`. At the top-level, the software development process simply consists of a series of releases of a particular application. The release operator has sub-goals requiring that the application in question be built and tested. These sub-goals express decomposition of the release process into the sequence of building and testing the application. Decomposition ends when a operator corresponds to a primitive action such as compiling a source file to produce an object file. These operators have no sub-goals and are instead associated with the actual commands required to carry them out.

Grapple works in two modes: plan creation and plan recognition. The programmer may ask Grapple to satisfy a particular goal. Grapple then attempts to formulate and execute a plan which achieves that goal. Of course, it may not possess enough information to complete execution of the plan itself. For example, the `release` operator requires that the application be tested. Testing of the application generally requires the intervention of a programmer. Such intervention causes Grapple to use plan recognition. Given that a top-level goal exists, Grapple examines the intended actions of the programmer and determines whether they are consistent with achieving that goal. If they are not consistent, Grapple can provide some indication of why this is so. Furthermore, Grapple can provide direction to the programmer as to what goals remain to be satisfied and, by displaying its plan to achieve these goals, assist the programmer in carrying them out. Thus, once the software development process has been codified for Grapple, it can both automate the process and provide assistance to programmers working in the context of that process.

Clearly, Grapple encompasses a wider scope of configuration management issues than the systems surveyed earlier in this chapter. The requirements for identification, control, specification, monitoring and repair are all addressed to a degree. Grapple's major weakness is that it is, fundamentally, a passive system. Action is initiated by the programmer requesting satisfaction of a goal. Upon achievement of that goal, Grapple becomes inactive. Maintaining relationships on an ongoing basis is not possible. The root of the problem is that monitoring and repair are not coupled in Grapple. While Grapple can access the state of the system, it does not actively monitor it. Commands issued by the programmer are inspected only when satisfaction of a goal has been requested, which is itself a programmer-initiated action. Thus, while Grapple possesses the knowledge of how to repair a system, it does not initiate repair.

The other weakness of Grapple is that it is a prototype system. It is not integrated into an actual development environment, which means that issues such as non-interference, integration, performance, and distribution have not been examined in detail. For some of these issues there are obvious concerns. For example, planning of the type used in Grapple, has been shown to be NP-Hard [10]. Also, Grapple was not designed as a distributed application, so distribution would entail a wide variety of changes, not the least of which would be adopting a distributed planning paradigm.

3.3.2 The Conic System

In Grapple, configuration management tools are fitted to an existing environment. Another avenue of research is to design a system in which configuration management concerns are addressed from the outset. One such system is Conic [30][32][34]. The emphasis in Conic is on the construction of applications out of simpler components. At the lowest level, these components are programmed in a derivative of the Pascal language. The novel feature of Conic is a separate configuration language which allows a programmer to specify a system in terms of the components of which it is comprised, the locations within the system where these components should exist, and the patterns of communication among the components. The system is hierarchical, in that a constructed system may be used as a primitive element within another. Conic also possesses facilities for monitoring applications and for modifying their composition, location, and patterns of communication.

The creators of the Conic system emphasize its communication model. The basis of the model is the inter-object link. A link has the following properties:

(1) It is typed. The information which it carries must be of a particular type.

(2) Transmission of information on the link is independent of the identity of the receiver. The link has a local name in the domain of the sender. Reference

to the link by the sender is always to this local name regardless of the identity of the object to which the link is connected.

(3) Reception of information from the link is independent of the identity of the sender. The link has a local name in the domain of the receiver. Reference to the link by the receiver is always via this local name regardless of identity of the object to which the link is connected.

(4) The links of two objects may be connected to one another to establish a communication path between them. This connection can be performed externally to the objects being connected. Connection of two objects via their links requires only that the links be type-compatible.

Properties 2, 3, and 4 allow the development of configuration-independent software. Because inter-object links are referred to only via a local name, it is not possible to embed a given communication structure into a set of objects; there is no means of directly addressing the objects forming this structure. Although this may seem unusual, programmers of Unix filters have been practicing the same type of configuration independence for years. A typical Unix filter reads from its standard input and writes to its standard output. The filter program contains no direct references to other processes or programs. Instead, the user is able to configure a collection of filters into a particular application by using Unix's piping capability to link the filters to one another.

Although such inter-object links are not necessary for configuration management, they enhance its usefulness greatly. Magee et. al state [34]:

> The Conic task module, thus, provides configuration independence in that all references are to local objects and there is no direct naming of other modules or communication entities. This means there is no configuration information embedded in the programming language... It is difficult to envisage how arbitrary changes could be incorporated into a system where such configuration information and control is embedded in the programming language.

Because configuration information and control is not embedded in the programming language, it must be represented by some other means. Another important feature of Conic is its separate configuration language. This separation is consistent with the assertion that only separation of programming-in-the-small from programming-in-the-large results in the creation of modular components suited for re-use. The reasoning behind this assertion is similar to that expressed by Magee et. al. in explaining the importance of configuration independence. If one understands programming-in-the-large to refer to the manner in which software components are structured and related to one another, then the more configuration information which is embedded in a given software component, the fewer situations exist which are consistent with that embedded information. Separate languages for component programming and configuration minimize embedded configuration information and, hence, maximize flexibility.

Conic provides an excellent framework for the development of configurable software. Tools for management of the applications created in this environment are less well-developed. The Conic configuration language is used only when a system is instantiated. While facilities are provided for accessing and modifying existing systems, the applications which use these facilities are simple and programmer-driven. In effect, Conic only addresses the requirements for identification, control, and, in a limited manner, specification.

Recent work has resulted in a more comprehensive configuration management tool for the Conic system. Andrew Young's dissertation [58] details the design and implementation of a system which maintains the structural integrity of Conic applications. His system allows the specification of constraints governing the interconnection of components. Configuration changes to a system are validated with respect to this specification, and changes violating the constraints are disallowed. Failure of a component causes invocation of a fault recovery manager which attempts to restore the system to a valid state.

Young makes the interesting assertion that "management will use only structural information, information about internal application states will not be available." [58] Thus, his system is constructed on the assumption that only constraints over inter-object links are required. This allows him to create a formal model of the system, and makes possible the determination of whether specifications are in conflict with one another and other such useful properties. However, he does not provide any support for the assertion, and it is easy to demonstrate circumstances in which structural information alone is not sufficient. For example, the temporal constraints maintained by `make` are not structural. Even the structural constraints of SySL are not structural according to Young's definition, because in his context "structure" refers only to the pattern of interconnections.

Despite this limitation, Young's work presents several novel features. Using a logical formalism to express configuration information makes it possible to reason about configuration management. For example, for many configuration specifications, the repair process can be completely automated because it is possible to mechanistically determine how to respond to a given failure. Monitoring and repair are closely coupled, and Young's system is the first which we have looked at in which faults are automatically detected and subsequently repaired. Because Conic focuses on providing an environment for dynamic configuration, the requirements for non-interference and integration are well handled. Conic is, furthermore, a distributed system, so by default, Young's system provides some level of support for distributed configuration management. However, the system uses Prolog for both recording and manipulating configuration information. There is no discussion of how the workload of the system could be shared among several incarnations of the interpreter and so one must assume that currently this aspect of the system is centralized. Finally, all of the examples are presented as though the system contained only a single configuration. There is no discussion of how configuration specifications are identified and mapped on to actual applications.

3.3.3 The Network Management System

The last system we examine is the Network Management System (NMS) [57]. NMS is another advanced configuration management system capable of monitoring and automatically repairing configurations. In NMS, the close coupling between monitoring and repair is achieved by using active database technology along with a data manipulation language which makes it easy to extract useful information about the behaviour of the network. It is worthwhile to note at this point, that while NMS is presented as a network management tool, there is no reason that the basic approach could not be used in a more generic configuration management setting.

At the core of NMS is a database containing information about the state of the network. NMS assumes that the network is instrumented in some manner and that the database is updated when elements of the network change state. Furthermore, it assumes that operations on entities in the database cause corresponding actions to be performed on the elements of the network. In essence, the database contains proxies for the managed elements of the network. The data management language of NMS allows additional inferred data to be added to the database. For example, a trace of the values of a particular attribute of a network device may be created. This trace could then be used to compute average values for that attribute over various intervals. The language also possesses a number of temporal operators allowing the definition of events. One of the basic events is simply calendar time. More complex events can be described in terms of the conjunction of simple events, the conjunction of events with existence of a particular tuple in the database, and intervals between events.

Added to this data manipulation language is a language for defining a rule database. The rule database allows automated repair. Each rule has a set of conditions which are required to trigger it, and a set of actions which it performs when it is triggered. The actions consists of adding and deleting tuples from the database. Of course, since some elements of the database are proxies for

network components, the addition or deletion of tuples can effect actual changes in the network. Addition and deletion of tuples may also lead to the activation of further rules, so a degree of hierarchical structuring is possible. The rule database, thus, represents knowledge about how the network should be managed.

Clearly, NMS relies heavily on the functionality of the underlying database. For example, if the database is distributed, then NMS is a distributed network management system. Consequently, many of the requirements for a configuration management system become requirements for the active database used. NMS does not appear to have been implemented and it is difficult to assess the practicality of an implementation. Since distributed active databases are still very much a research issue, it seems safe to assume that the problems would be non-trivial.

Even assuming the existence of a suitable database, NMS is not an ideal system. By effectively replicating the network state in the database, NMS adds a layer of overhead not present in a system which directly manipulates the components of the network. Furthermore, although NMS allows arbitrary relationships to be expressed via the database, it does not maintain constraints on those relationships directly. The rule database can be used to implement a system whereby such constraints are maintained, but the constraints themselves would be implicit. For example, if the database maintained information on the date of last modification of various files, then rules could be installed which compared these dates and re-compiled files when necessary. In other words, the functionality of `make` could be duplicated, but the actual constraint that the date of one file needs to be greater then another would be implicitly rather than explicitly stated. Because constraints are implicit in NMS it is difficult for these constraints to be automatically manipulated and analyzed. Furthermore, understanding of complex configurations is difficult because the configuration information in NMS tells us *what* management activities will occur, but the *motivation* for having these activities occur must be deduced by the user. It is interesting

to note that Young's configuration management system for Conic allows explicit expression of constraints in a restricted manner, that is, only communication linkages between components may be constrained, while NMS permits complex relationships involving any observable attributes of a component, but does not provide for constraining those relationships.

3.4 Summary

A brief review of a sample of configuration management systems cannot provide conclusive proof that the problems of configuration management have not been solved. However, the systems reviewed in this chapter were chosen because they typify current approaches to configuration management and consequently reveal typical shortcomings. For example, some systems [2][21][49] provide the basic environment for constructing distributed applications, but have only weak support for any sort of management. Others [20][43][11], are targeted for very specific domains; they lack a general model of configuration management. The Darwin system [36] has a model of management, but appears to be difficult to implement efficiently in a distributed system. Lack of active management is also a common problem [41][48][52].

An important observation about existing configuration management systems is that *none of them provide both the ability to express arbitrary relationships between components and the ability to constrain those relationships.* As was pointed out in Chapter 1, *the essence of configuration management is expressing and constraining the relationships between the components of a configuration.* A complete implementation providing this capability presents interesting problems with respect to monitoring, integration, performance and distribution. Most current systems do not address configuration management in a general manner. Most existing configuration management tools are targeted for specific domains: software engineering, network management, communications structuring, etc. Yet, the tasks performed in all of these cases can easily be seen as expressions of the same underlying paradigm, namely constraining the relationships of com-

ponents to one another. No other research has investigated the possibility of performing all of these types of configuration management with a single formalism.

There are three *major* contributions of the research presented in this thesis. The first is the development of this new model of configuration management which emphasizes the need for a general formalism, recognizes that configuration management must be an active process, and ensures that configuration management suited to a distributed environment. The second is the design of a new configuration language based on this model. The significant features of this new language are that it provides an object-oriented approach to configuration management, a flexible mechanism for identifying objects which are to be managed, and a constraint-based language for describing the relationships between objects. The third and final area in which significant research contributions have been made is in providing an implementation of this new configuration language. This implementation establishes the viability of both the RCMS model and the configuration language. Also, the implementation establishes the sort of services required in order to implement a general, active, and distributed CMS. Finally, several interesting implementation techniques were used or proposed in order to address issues such as efficient evaluation of quantified expressions, preserving consistency through locking, improving the performance of component set computations, and improving the level of distribution of management activities

The remainder of this thesis describes the design and implementation of the RCMS. It presents novel features in terms of its model of configuration management, the implementation techniques used, and the practical knowledge gained through experimenting with the completed system. Although the RCMS does not represent the ideal configuration management system, it makes significant contributions to research into configuration management, adding to the base

of knowledge acquired through the design and implementation of systems such as have been examined in this chapter.

CHAPTER 4

The Raven Configuration Management System

The Raven Configuration Management System (RCMS) is unique. It is a general system which actively manages configurations in a distributed environment. It alone addresses the nine requirements for configuration management described in Chapter 2. This chapter describes how to use RCMS. However, because RCMS has been implemented within the Raven environment, the first step of this process is, necessarily, a brief review of Raven.

4.1 The Raven System

The Raven System [39] is a project at the Computer Science Department of the University of British Columbia. The intent of the project is to explore a variety of issues in distributed object-oriented computing. The system currently consists of a compiler, class library, and set of runt-ime routines which provide a distributed computing environment running under the Unix or Mach operating systems. The programming language for the system is called, not unexpectedly, Raven.

4.1.1 The Raven Programming Language

Although Raven is an object-oriented language, it bears a fairly substantial syntactic resemblance to C [28]. The fundamental unit of programming in Raven is the *class* and the fundamental unit of allocation is the *object*. Each object is an instance of a class. A class definition in Raven consists of two parts. First is the interface declaration which describes both the structure of an instance of that class and the operations which may be performed on such an instance. The structure of an instance consists of a set of instance variables which maintain its state. So, for example, we can conceive of a class named "Stack" whose instances are incarnations of that venerable data structure. To simplify matters somewhat, the stack is assumed to store integers. In Raven, the interface declaration for such a class might look like:

```
class Stack
    {
    contents : Array[Int];
    top : Int;
    behav Push(val:Int);
    behav Pop() : Int;
    }
```

The first part of the interface declares an instance variable `contents`, used to maintain the values stored on the stack. Its type, `Array[Int]`, indicates that it is an array of integers.[1] Note that type declarations are more Pascal-like than C-like in Raven; the name of an item is followed by a colon and the type which that item has. The instance variable `top` is an index into `contents` indicating where the top of the stack is. Next, are two behaviour declarations which define how a programmer may manipulate an instance of `Stack`. As might be expected, an instance of `Stack` supports two behaviours, one to push a value on to the stack and another to pop a value off.

1. `Array[Int]` is an example of a parameterized type. Raven provides a general-purpose mechanism for the definition of parameterized types. Type parameters are specified as a list of type names enclosed in brackets following the primary type name.

The second part of a class definition is a set of behaviour definitions which constitute the implementation of an instance of the class. The behaviour definitions describe how an operation like `Push()` or `Pop()` changes the state of an instance. The implementation for `Push()` might look like:

```
behav Push
    {
    contents.atPut(top, val);
    top += 1;
    }
```

The instance variable `contents` is, in fact, simply a reference to another object. These references are called capabilities. `Contents` refers to an instance of the class `Array[Int]` and, accordingly, it may only be operated on via the collection of behaviours defined for that class. One of these behaviours is `atPut()`, which places the value supplied as its second argument at the location in the array indicated by its first argument. The expression:

```
contents.atPut(top, val)
```

is called an invocation, and it means that the `atPut()` operation should be performed on the object referred to by `contents`, using `top` and `val` as parameters.

The important feature of Raven (and, in fact, any object-oriented language) is that each object has a well-defined interface and the only way to modify the state of an object is via that interface. For a simple object such as `Stack`, that interface consists of only the behaviours found in the class interface declaration. Raven has several other means for defining interfaces. For example, if an instance variable declaration is followed by the keyword `public`, then that instance variable becomes part of the exported interface of that class. Users of instances of that class may then directly access the instance variable.

4.1.2 Inter-Object Links in Raven

One of the most important interface features that Raven offers is the inter-object link. Conceptually, these are very similar to the links found in Conic [30]. An inter-object link is one-half of a typed, unidirectional communication channel. Links come in two flavours, inbound links and outbound links. When an outbound link is connected to an inbound link, a complete communication channel is established. The links which an object possesses are part of the interface of that object. A simple data filtering class might have the declaration:

```
class SimpleFilter
    {
    link in dataIn(data:Int);
    link out dataOut(Int);
    }
```

This defines one inbound link called `dataIn` and one outbound link called `dataOut`. The inbound link accepts data of type `Int` and the outbound link transmits data of type `Int`. For each inbound link, there must be a corresponding implementation which is executed when a data item arrives on that link. So, we might supply an implementation for `dataIn` looking like:

```
link dataIn
    {
    if (data > 0) send dataOut(data);
    }
```

The incoming data item is checked to see whether it is greater than zero. If so, it is sent on the outbound link, otherwise no action is taken. Instances of this class, thus, act as simple filters which remove all non-positive values from a data stream. Notice that the implementation for an inbound link is very similar to the implementation for a behaviour. In fact, the two are identical, save for the differing preamble, that is the keyword `link` rather than `behav`, and the fact that an inbound link never returns a value (because links are unidirectional). There is no implementation for an outbound link: it simply represents a communication endpoint to which data is sent. Both inbound and outbound links may participate in zero or more connections. A data value sent

on an outbound link connected to more than one inbound link is sent to all of those inbound links. An inbound link connected to more than one outbound link receives values sent on any of those outbound links. The Raven runtime provides services for connection/disconnection of inbound and outbound links as well as several other utility functions such as counting the number of connections emanating from an outbound link or arriving at an inbound link.

Just as in Conic, links do not explicitly identify the endpoints of the communication channel. Within an object, a link is referred to only by its local name. Inter-object links, therefore, permit the development of configuration independent software in Raven. Traditional operation invocation and linked-based communication complement each other. Operation invocation, in which the target of the operation is explicitly specified, permits the programmer to tightly bind objects to one another. Link-based communication permits looser bindings because it eliminates explicit identification of the target of an invocation and allows external control of the binding.

4.1.3 Other Features of Raven

There are several other features of Raven which are relevant to an understanding of RCMS. Invocation in Raven is location transparent. As noted above, objects are identified via references which are called *capabilities*. A capability may refer to an object on the local machine or on a remote machine, and the programmer need not be concerned with the distinction. Raven also provides support for concurrency control, recoverability, and persistence for objects. The manner in which these features are used is examined in more detail in Chapter 5, which explains how RCMS is implemented. An important point to make at this juncture is that while these features have eased the task of implementing RCMS, the approach taken in RCMS does not require these services. Section 6.6 provides information about how the concepts of the RCMS and even large portions of the RMCS implementation can be used in other environments.

4.2 The Raven Configuration Management System

The remainder of this chapter provides a general explanation of RCMS via the presentation of a number of examples. These examples highlight the various features of RCMS and demonstrate how the configuration management model described in Chapter 2 is implemented in RCMS. The basis of the implementation is a new programming language for describing configurations and configuration information. The grammar for this language and a more formal treatment of its semantics appear in Appendix A.

A central insight of RCMS is that object-oriented programming and configuration management are really not so different. In programming, the implementation of any given object binds together other objects and primitive values to support a particular set of operations. The glue which holds it all together is the code which implements each operation. The task of a configuration management system, as shown in Chapter 2, is to bind together a set of components and maintain relationships between them. In RCMS, the glue which holds it all together is a specification of the relationships, constraints, and repair information. Configurations are actually embodied as objects and are, in fact, called *composite objects*. The more traditional objects, programmed in the Raven language, are called *elementary objects*.

Because the Raven system itself is uniformly object-oriented, it is straightforward for the implementation of RCMS to conform to the general nature of the model described in Chapter 2. The combination of full access to Raven's elementary objects and the ability to use first order predicate logic provide RCMS with a powerful mechanism for expressing configuration management information.

Externally, composite objects differ very little from elementary objects. They have a well-defined interface. They can be created and destroyed. References to them can be included in

other composite or elementary objects. Indeed, the whole range of tools which exist for manipulating elementary objects also apply to composite objects. This is the first feature of RCMS which helps cleanly integrate configuration management into the Raven system.

Internally, composite objects differ substantially from elementary objects. The composite object is the manner in which RCMS represents a configuration. The model of configuration management presented in Chapter 2 thus demands that the definition of a composite object:

(1) identify the components of the configuration,

(2) define the relationships between the components,

(3) specify the constraints on those relationships,

(4) specify how to repair the configuration.

As was noted earlier in this section, RCMS uses first order predicate logic as its primary means of expressing configuration information. Sets which identify the components of a composite object can be specified by explicit enumeration or by predicates. Relationships between objects are also expressed as predicates, and the constraints on those relationships are simply expressions in predicate logic.

4.2.1 Defining Composite Objects

Each composite object is a member of a *composite class*. The fundamental unit of programming in RCMS is, thus, the composite class definition. A composite class definition begins with an *interface declaration* much like that of Raven. It consists of instance variable and link declarations. Here is a simple example:

```
composite ToyComp
    {
    f1, f2 : SimpleFilter;
    link in dataIn(p1:Int);
    link out dataOut(Int);
    }
```

Except for the initial keyword `composite`, this is syntactically identical to a Raven class interface declaration. This was intentional. It allows the Raven compiler to accept interface declarations from RCMS and vice-versa. A Raven programmer wanting to make use of an RCMS composite object can simply include its interface definition in his program and link his code with the class library which supplies its implementation. RCMS programmers can access elementary objects in a similar manner.

The interface for a composite class actually serves two purposes. First, as in Raven, it defines the interface of instances of that class. Second, the instance variables of a composite class serve to define objects which are the components of the configuration. In RCMS, a configuration and a composite object are the same thing. For example, the above example specifies a configuration of two objects of type `SimpleFilter`.

The existence of link definitions for a composite may seem puzzling. Since there is no imperative language in RCMS which could serve to implement an inbound link or provide means of sending data to an outbound link, it would appear that the presence of link declarations conflicts with the nature of RCMS. Links serve a somewhat different purpose in RCMS. They act as gateways between the system outside of a composite object and the components within it.

Externally, a link on a composite object is just like a link on an elementary object. Inbound links receive data and outbound links provide it. However, each inbound link declared within a composite implicitly creates an outbound link of the same name accessible only to its components. This outbound link may be connected to the inbound link of any component. Once such a connection is created data sent to the inbound link of the composite appears on the inbound link of the component. A similar situation holds for outbound links. Thus, the interface of the composite can be mapped onto the interfaces of its components without the need for an imperative programming language.

It should be noted that behaviour declarations are not permitted as part of the interface for a composite class. The support of behaviour declarations for a composite class requires a similar mechanism for mapping incoming invocations onto the interfaces of components objects. The syntax and semantics of such a mechanism have been developed, but implementation was not considered essential for a proof-of-concept system. Inter-object links provide a satisfactory mechanism for controlling objects, and they are a flexible configuration mechanism, allowing the communication structure of an application to be easily changed at run-time. Thus, even without behaviour declarations for composite objects, the initial version of RCMS possesses adequate functionality.

4.2.2 The Specification Clause

Following the declaration of the interface for a composite class, the similarities between Raven and RCMS disappear. This is appropriate. Composite objects and elementary objects are used in the same way and it is natural that their interfaces be specified similarly. However, composite objects are a means for achieving configuration management in an object-oriented environment and so the implementation of a composite object must be in terms of relationships.

The next element of a composite class definition is a *specification clause*. The specification clause allows relationships to be defined and constraints to be stated. In RCMS, relationships are expressed in terms of predicates. For example, in a software engineering setting one might like to have a relationship called `DependsOn` indicating a dependency from one module to another. Such a relationship can be expressed as a predicate `DependsOn(A,B)`, which is true if module `A` depends on module `B`. Predicates, in turn, are defined in terms of logical operators and the externally visible interfaces of objects. If the attributes of the objects involved are suitable, the predicate may be quite trivial. If the objects representing software modules, for example, pos-

sessed an externally visible instance variable called `dependsOn`, then the following RCMS definition would suffice:

```
define DependsOn(A,B) <- A.dependsOn == B;
```

RCMS supports all of the usual logical operators, universal quantification, existential quantification, and a few special set operators. A significant restriction on the definition of predicates and constraints is that they may not invoke behaviours which have side-effects, that is, behaviours which alter the state of the object on which they are invoked. The motivation for this restriction is that the act of computing whether a composite is in an acceptable state should not cause the system to be modified. Predicates and constraints may observe the states of objects, but they may not alter them. The only time at which RCMS alters the state of an object which is a component of a composite is when it is performing repair on that composite. Specification of repair information is detailed in Section 4.2.5.

4.2.3 Sets in RCMS

Sets are generally valuable when using logic and RCMS provides several mechanisms to assist the programmer working with sets. However, before presenting them, a minor digression into the semantics of sets in Raven is needed. Raven supports two varieties of set classes. Both classes have the same interface. They differ with regard to the semantics of object equality. Class `Set` uses an equality operator which is usually referred to as *deep equal*. Two objects are deep equal only if they are members of the same class, and the values for each of their instance variables are also deep equal. The only primitive values in Raven are integers, and two integers are said to be deep equal if they have the same value. Class `IdSet` uses object identity as its equality operator. Two objects are identical if they are, in fact, the same instance of a particular class. Identity implies deep equality, but not vice-versa. For example, in Raven there may be many instances of the class `String` which contain the text "Hello World." All of these

instances are deep equal to one another. However, any distinct pair of these instances are not identical. Suppose that six of these instances were added to an instance of class `Set`. The resulting size of the set would be one because sets can only hold one element of a particular value. If the same six instances were added to an instance of class `IdSet`, the resulting size would be six. Since the six instances are not identical, the `IdSet` treats them as separate values.

In the realm of configuration management, `IdSets` tend to be more useful than `Sets`. One constructs configurations out of distinct objects, and it is usually undesirable to lose the distinction between two objects simply because they represent the same value. Consequently, when RCMS creates sets it uses instances of `IdSet`.

RCMS creates new sets at run-time in response to the use of set constructors within a logical expression. RCMS provides two varieties of set constructors. In the first, the members of the set are explicitly enumerated. Suppose that `c1`, `c2`, and `c3` were components of a composite object, then the expression:

```
{c1, c2, c3}
```

is an object of type `IdSet` which contains references to those three components. The second variety uses a logical expression which acts as a membership predicate for the set. For example, suppose that we have a configuration composed of software modules. Each module supports a boolean attribute called `tested`. Then the expression:

```
all X : X.tested
```

is an object of type `IdSet` containing references to the software modules whose `tested` attribute is set. Clearly, the set constructed in this example should contain only software modules which are a part of this composite and not all software modules anywhere in the system which have their `tested` attribute set. To achieve this behaviour RCMS does not look at all

objects in the system when constructing a set, but rather, restricts itself to examining the contents of the *change set* of the composite. Intuitively, the change set of a composite consists of all objects which could have an effect on the state of the composite. That is, if a change to an object X results in an observable difference in the behaviour of a composite, then X is a member of the change set of that composite.

To formally specify the change set, we start with the assertion that the state of any Raven object is defined solely in terms of its instance variables. Instance variables hold values of certain primitive types, for example integers, or capabilities for objects. The state of an object is a function of the primitive values stored in its instance variables and the states of any objects for which it has capabilities. The reason for the latter condition is that the value returned by an invocation on an object may be computed in terms of the primitive values of its instance variables and the values returned by invocations on objects for which it has capabilities. Thus, the change set of a particular object X is defined recursively as the union of the set of objects whose capabilities are stored in the instance variables of X and the change sets of these same objects. In effect, the change set of X consists of all objects reachable from X by following the object references stored in instance data. Finally, the change set of a composite, is just the union of the change sets of its instance variables.

Many data structures represent aggregates. Arrays, sets, symbol tables, etc. are all composed of a number of structurally similar items. Frequently, it is desirable to be able to deal with each element of the aggregate in turn. In Raven, this is possible through a process called *iteration*. All classes which represent aggregate data structures support a common set of behaviours which allow the programmer to access the elements of the aggregate one at a time. Generally, this access is done in the context of a loop. A loop, unfortunately, is an imperative control structure

and as such, not available within RCMS. Nonetheless, the need to iterate over aggregates still exists.

The solution to this problem is the `in` operator. It is a binary predicate which uses infix notation. The expression `A in B` is true, iff iteration over the elements of `B` yields an object which is identical to `A`. If one thinks of an aggregate object as being a set of objects, then the `in` operator is simply a set membership test. The power of the `in` operator lies in the fact that one can use it to alter the *domain of quantification*, that is, the set of objects used when evaluating a quantified expression. Normally, the domain of quantification is the change set of a given composite. However, the domain can be specified explicitly using the `in` operator. For example, suppose the component `destinations` is an array of routing information, then the expression:

```
for all X [X in destinations] Reachable(X)
```

evaluates to true iff each of the objects in `destinations` satisfies the predicate `Reachable()`. The expression in brackets is called a *filter expression.* It may be any logical expression, but used in this way it modifies the domain of quantification, for both universal and existential quantification, to be the set of objects contained in the aggregate specified on the right side of the `in` operator.[1] Quantification, thus, effectively functions as iteration in RCMS.

4.2.4 Constraints on Relationships

The specification clause contains not only predicate definitions, but also the core of the composite class definition: the *constraints.* Once again, this derives from the model presented in Chapter 2, which emphasized the importance of constrained relationships as the basis for configuration management. A constraint is a logical expression. A composite is *valid with respect to* a particular constraint if that constraint evaluates to true. A composite is *valid* if it is

1. The filter expression in a quantified expression is permitted to be any logical expression. The full semantics of the filter expression are provided in Section A-4.2.

valid with respect to each of its constraints. The purpose of RCMS is to ensure that all instances of composite classes are valid. The process of evaluating the constraints for a particular composite to determine if it is valid is referred to as *validation*. Returning to the simple example given earlier in this section, its specification clause might look like:

```
specification
    {
    f1 = SimpleFilter.new(0);
    f2 = SimpleFilter.new(1);
    1: Connected(self, dataIn, f1, dataIn);
    2: Connected(f1, dataOut, f2, dataIn);
    3: Connected(f2, dataOut, self, dataOut);
    }
```

The first two constraints make use of special equality operators which were found to be required in order to allow composites to be properly initialized and maintained. There are, in fact, three different equality operators in RCMS represented by the symbols ==, :=, and =. The first, called *equals* (==), is an ordinary infix predicate which indicates whether its left and right sides are identical to one another. The second and third are more akin to assignment operators. The second, called *temporal assignment* (:=), ensures that changes in the value on its right are always propagated to the instance variable on the left. The third, called *initial assignment* (=), assigns the value on its right to the instance variable on the left, but only when the composite is first created. Thus, the first two constraints of the example create two filter objects and assign them to the instance variables of the composite. Note that initial assignment and temporal assignment depart from the norm for RCMS slightly because they effectively combine specification and repair information. That is, both types of assignment not only detect the violation of a constraint (namely that the values on their left and right hand sides differ), but also establish a repair mechanism which is used to satisfy the constraint, that is, assignment of the value on the right hand side to the variable on the left hand side.

The predicate `Connected()` is built-in to RCMS. It is actually just syntactic sugar for an ordinary object invocation which indicates whether a pair of links are connected to one another. Its arguments consist of two capability/link name pairs. The first pair indicates an outbound link and the second an inbound link. As was noted in the discussion of links, the declaration of an inbound link for a composite object implicitly creates an outbound link of the same name. That outbound link is accessed by using the built-in variable `self`, which holds a capability for the composite object. Thus, the first constraint, `Connected(self, dataIn, f1, dataIn)`, indicates that the implicit outbound link `dataIn` should be connected to the inbound link `dataIn` of component `f1`. In this manner, data sent to link `dataIn` of the composite is forwarded to link `dataIn` of component `f1`. The result of the last three constraints of this composite is that the composite object should have structure like that shown in Figure 2.

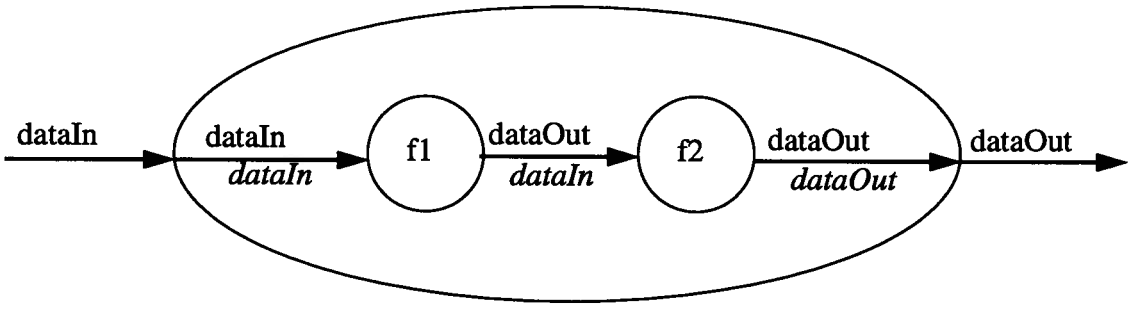

Figure 2 – Link Connections in a Simple Composite

4.2.5 The Repair Clause

The set of constraints in the specification clause of a composite class allow RCMS to determine when a composite is valid. In order for RCMS to conform to the active model of configuration management presented in Chapter 2, it must also be able to restore a composite to a valid state when changes to its components have caused a constraint to be violated. The third portion of a composite class definition provides information about how RCMS carries these repairs.

Two basic repair strategies are used in RCMS. One relies on Raven's support for recoverable objects and the other provides the RCMS programmer with a limited imperative repair language. The decision to provide an imperative language was motivated principally by expediency. The imperative repair language has allowed RCMS to be designed, implemented, and tested in a reasonable length of time. Fortunately, the use of an imperative repair language does not preclude the addition of more automated repair mechanisms. In his work with Conic [58], Young briefly notes that an expert system could be used to automate repair activities. Since RCMS specifications are declarative in nature, it should be possible to use an expert system for repair within RCMS as well. This issue is dealt with more fully in Section 6.5.

The two repair strategies offered by RCMS correspond to two different models of repair. RCMS can either prevent changes which cause the violation of a constraint, or take a configuration which is no longer valid and modify it so that it becomes valid. The former is *roll-back repair* and the latter is *roll-forward repair.*

4.2.6 Roll-back Repair

Roll-back repair is specified by a recovery block which consists simply of the keyword recoverable:

```
repair
    {
    recoverable;
    }
```

For roll-back repair to be possible all of the components of the composite must be recoverable. Recoverability in Raven means that at any point in the execution of a behaviour of a recoverable object, both that object, and any recoverable objects which have been modified either directly or indirectly by the execution of that behaviour, can be restored to their original state. As a thread of control executes within Raven it enters and exits objects as a result of performing invoca-

tions. When a behaviour is invoked on a particular object, that object is entered. When that behaviour returns control to the point of invocation, the object is exited. Nested invocation result in sequences of objects being entered, then exited. If one looks at the set of objects which have been entered by a particular thread, either none of the objects is recoverable, or there is a first object which is recoverable. Upon entering this first recoverable object, the thread is said to have entered a top-level recovery block. Exiting from the object corresponds to exiting the top-level recovery block. A recoverable composite is validated upon exit from a top-level recovery block of any of its components. If the changes made during the recovery block cause the composite to be invalid with respect to its specification, then all changes made within the recovery block are rolled-back.

4.2.7 Roll-forward Repair

Roll-forward repair is specified by a repair clause containing repair scripts. These scripts are written in an imperative language which provides a subset of the control features of the Raven programming language [39] merged with the language for logical expressions used by RCMS. Each constraint of the specification clause may be preceded by an integer label. Each repair script of the repair clause may be preceded by a similar label. When evaluation of a particular constraint yields *false*, the corresponding repair script is executed. After execution of a repair script, the entire composite is re-validated. Thus, once RCMS has detected that a composite has changed state it repeatedly validates and repairs the composite until the composite becomes valid. This is referred to as the *validation/repair phase*. Because repair scripts may contain arbitrary invocations on components of the composite, it is not possible to verify in advance that the repair/validation phase will terminate. It is the responsibility of the RCMS programmer to ensure that repair scripts interact in such a manner as to eventually produce a valid composite.

The control constructs allowed in a repair script are the while loop, the do loop, the for loop, and the if-then-else statement. Variables may be declared and initialized and all of the components of the composite are accessible. There are two built-in variables defined. The variable `self` is a reference to the composite object. The variable `system` is a reference to an object which provides an interface to various Raven primitives such as the connecting and disconnecting of inter-object links. So, for example, a repair clause for the simple composite object defined earlier in the chapter is:

```
repair
   {
   1: system.makeLink(self, dataIn, f1, dataIn);
   2: system.makeLink(f1, dataOut, f2, dataIn);
   3: system.makeLink(f2, dataOut, self, dataOut);
   }
```

4.3 A Comparison of the Model and the Implementation

Now that the basic features of the RCMS language have been presented, we can stand back and evaluate the correspondence between the concrete system, RCMS, and the model of management, presented in Chapter 2, on which it is based. As noted in Section 3.4, the most significant aspects of the model are that it is general, active, and distributed.

The generality of the model stems from the uniformly object-oriented view of the universe of entities to be managed, combined with the power of relationships and constraints as a means of controlling the interactions between those entities. By working within the Raven system, which is uniformly object-oriented, RCMS gains the required uniformity. The specification clause of a composite class definition provides a locus for expressing relationships and constraints, thus, meeting the second requirement of generality.

The active nature of the model sprang from the combination of several requirements: specification, repair, and monitoring. An active configuration management system monitors the status of

the system in real-time, compares the states of each configuration to their corresponding specifications, and intervenes to restore configurations to valid states if they do not meet their specifications. The specification and repair clauses of a composite class definition, combined with run-time monitoring techniques which have been added to the Raven system, make RCMS an active configuration management system.

The model is suited to a distributed computing environment because configuration information and capabilities are closely bound to configurations. There is no centralized database of configuration information, nor a centralized configuration manager which controls configuration activities. RCMS conforms to this aspect of the model by translating the concept of a configuration into a concrete entity within the system: the composite object. Component identification, predicate definitions, constraints, and repair information are all directly stored in the composite object. Each composite object is, in fact, a configuration management agent, responsible for managing the interactions between its components. By distributing composite objects in the system, one naturally distributes the task of configuration management.

4.4 Examples

At this point, the basic features of the RCMS language have been covered and the overall design of the implementation has been compared to the model of configuration management presented in Chapter 2. The remainder of this chapter is devoted to the presentation of four somewhat more complex examples which should provide a better understanding of practical uses for RCMS. Some additional features of RCMS will be introduced along the way. Readers wishing a comprehensive and more structured description of the language are referred to Appendix A.

4.4.1 A Directory Tree

There are two styles of programming that one might use with RCMS which differ with regards to whether the components of the composite object are externally visible. In some circumstances it is desirable to group a collection of objects together and provide certain constraints on their relationships, but still permit access to the individual objects. In other situations, a composite object should be treated more as a black box, accessed only through its inter-object links. The delineation between these two approaches is not absolute; it maybe desirable to hide some objects and expose others. In RCMS, as in Raven, a component is made externally visible by using the keyword `public` on the instance variable declaration corresponding to that component.

The first full example of this chapter is a composite object whose components remain externally accessible. The composite is a simple directory tree, and RCMS is used to ensure that no loops exist in the tree. The directory tree is composed of elementary Raven objects of type `DirNode`. Each node may hold an arbitrary number of entries and each entry maps a textual name to a reference to another object. A tree is built up by having an entry in one `DirNode` which maps to another `DirNode`. Entries which map to any other sort of object are leaves. The interface for class `DirNode` is given in Figure 3. The important feature of `DirNode` is that it supports the

```
#include <Basic.r>
#include <Iterator.h>

class DirNode recoverable
    {
    behav addEntry(name:String, item:Object);
    behav remEntry(name:String);
    behav lookup(name:String) : Object;
    behav getIterator() : Iterator[Object];
    }
```

Figure 3 – Interface for a Raven Directory Node

`getIterator()` behaviour. This is the behaviour which an object must support in order for the `in` operator of RCMS to work. Iteration over a `DirNode` produces the set of objects which are stored in it, but not their names.

In order to make manipulation of the directory tree straightforward, all of the `DirNodes` in the tree are externally visible. If they were not, then the composite would have to provide an interface for adding, deleting, and looking up entries. Since these functions are already provided by the nodes of the directory tree, it is much more straightforward to simply allow the programmer to manipulate the tree directly.

Interface Clause

The complete definition for the composite is given in Figure 4. Several new features of RCMS are used. First, the definition of the instance variable `tree` is followed by the keyword `indirect`. This indicates that `tree` itself is not part of the composite, rather the contents of `tree` as obtained by iterating over it are the components. Use of the indirect attribute thus permits the specification of composites whose components are a dynamically changing set of objects. Without a construct such as this, a composite such as the directory tree could not be defined because there is no way to declare in advance a set of instance variables to hold the references to the nodes of the tree. The set of nodes changes over time as nodes are added to or deleted from the tree. The `indirect` attribute may only be used on instance variables which support iteration.

Another new feature in the interface of the directory tree is the definition of a `constructor()`. In Raven, a constructor is a special behaviour which is invoked by the system when a new instance of a class is created. It is used to initialize the internal state of the object. In RCMS behaviour definitions are not permitted and so the presence of a constructor definition would appear to be incongruous. However, composite objects may also require initialization and, in

```
#include <Basic.r>
#include <Composite.h>
#include <IdSet.h>
#include "DirNode.h"

composite Tree
    {
    root : DirNode public;
    tree : IdSet[DirNode] indirect;
    constructor(Root:DirNode);
    }

specification
    {
    define Path(X, Y) <- Y in X |
        there exists Z [Z in X] (Z.instanceOf() == DirNode) & Path(Z, Y);

    1: root = Root;
    2: tree := all X : X.instanceOf() == DirNode & Path(root, X);
    3: for all X [X in components] ~Path(X, X);
    }

repair
    {
    recoverable;
    }
```

Figure 4 – RCMS Definition For Directory Tree

particular, it is frequently desirable to parameterize them at the time of their creation. The constructor definition of a composite provides a convenient mechanism for such parameterization. There is no imperative code associated with a constructor in a composite, but the parameters declared for the constructor are accessible in the context of specification and repair clauses. For the directory tree, the constructor is used as a means to pass a root for the directory tree in to the composite. Thus, when a directory tree composite is created, the capability for the root of the directory must be passed in as a parameter. The first constraint of the specification clause is an initial assignment which stores this root `DirNode` into the instance variable `root`.

Specification Clause

The first few lines of the specification clause define a predicate `Path(X,Y)` which computes whether there is a path from directory node `X` to object `Y`. Currently, RCMS does not support type checking for predicates. Thus, there is no part of the definition which indicates the expected types of the arguments. Use of a predicate with the wrong type of arguments eventually generates a run-time error when an attempt is made to invoke an unsupported behaviour on one of the parameters. For example, if the object supplied as the first parameter to `Path()` is not a directory node, then a run-time error will likely result when RCMS attempts to iterate over `X` when it evaluates the `in` operator. `Path()` is recursively defined. The base case occurs when the object `Y` is contained within directory node `X`. For the recursive case, existential quantification with a filter expression of `Z in X` is used. Recall that this sort of filter expression causes the domain of quantification to be the set of objects returned by iterating over `X`. Since `X` is a directory node, the effect is to iterate over the entries of `X` checking to see whether there is a path from that entry (whose value is held in `Z`) to `Y`. Note the additional check to ensure that the entry bound to `Z` is itself an instance of `DirNode`, so as to guarantee that parameters of the correct type are passed to the recursive use of `Path()`. The behaviour `instanceOf()` is supported by all objects and returns a capability for the class which the object is an instance of.

The `Path()` predicate is used in both constraint 2 and constraint 3. Constraint 2 uses the temporal assignment operator and a set constructor to ensure that instance variable `tree` contains all the nodes of the directory tree excluding the root. Predicate `Path()` is used in the membership expression of the set constructor in order to identify when an object is reachable from the root of the directory. When changes are made to the directory tree, the set constructor is re-evaluated, producing an `IdSet` containing all of the nodes which are reachable from `root`. This set is then assigned to `tree`.

A set constructor such as that used in constraint 2 is like a quantified expression in that it is evaluated with respect to a particular domain. A set constructor effectively iterates over this domain evaluating its membership expression for each object. The value of the set constructor is an `IdSet` containing all those objects of the domain for which the membership expression evaluated to true. The default domain for a set constructor is the change set of the composite.

So, in constraint 2 the set constructor is evaluated by taking each object in the change set of the directory tree and checking, first, that the object is a directory node, and second, that there is a path from the root of the directory tree to that object. Once the value of this set constructor is assigned to `tree`, the components of the composite are precisely the nodes of the directory tree. Of course, additions to, or deletions from, directory nodes in the tree require that the set constructor in constraint 2 be re-evaluated in order to ensure that the set of components of the composite is kept up-to-date. This is the responsibility of the monitoring and repair facilities of RCMS.

In constraint 3, the `Path()` predicate provides the mechanism needed to achieve the goal of creating a specification which disallows loops in the directory tree. It is a simple constraint stating that no directory node may have a path to itself accomplishes this goal. In order for constraint 3 to have the desired interpretation, the domain of quantification for the universal quantifier must be the set of directory nodes in the directory tree. This, of course, is simply the set of components of the directory tree. RCMS provides a predefined identifier, `components`, which refers to an `IdSet` containing the current components of a composite. By using `components` in the filter expression of constraint 3 the correct domain of quantification is ensured.

Repair Clause

The repair strategy used for the directory tree is roll-back. This is indicated by a repair clause consisting solely of the keyword `recoverable`. Roll-back repair requires that all of the components of the composite support recovery. The directory tree is composed of a single type of component, instances of class `DirNode`, and as can be seen from Figure 3 the keyword recoverable is appended to its declaration. This guarantees that all instances of class `DirNode` are recoverable.

Updating and validation of a composite which uses roll-back recovery occurs upon exiting a top-level recovery block in which invocations on a component of the composite were performed. The validation procedure occurs in two phases. First, RCMS re-evaluates any constraints which could add components to or remove components from the composite. Then, all other constraints are evaluated.

In the directory tree composite, constraint 1 uses an initial assignment and so is only evaluated when then composite is created. Constraint 2 alters the value of `tree` which is used in defining the components for the directory tree. Thus, it is evaluated in the first phase of validation. That leaves only constraint 3 for the second phase of validation. If constraint 3 is not satisfied then the recovery block is rolled back. Note that if the recovery block contains invocations on objects other than the components of the directory tree, the effects of these invocations would be rolled back as well.

Figure 5 contains a sample interactive session showing a directory tree composite in use. The labels at the far right are not part of the interactive session, they simple serve as reference points for discussion. Starting at [a], several directory nodes are allocated and then connected together to create a simple directory tree as shown in Figure 6. At [b] the directory tree composite itself is

```
$ root = DirNode.new()                                    [a]
$ dir1 = DirNode.new()
$ root.addEntry("dir1", dir1)
...
$ tree = Tree.new(root)                                   [b]
$ listNames(root)                                         [c]
dir1, dir2
$ dir1.AddEntry("dir2", dir2)                             [d]
$ listNames(dir1)
dir2
$ dir1.addEntry("root", root)                             [e]
$ listNames(dir1)
dir2
```

Figure 5 – Sample Interactive Session with Directory Tree

created, parameterized by the root directory node. At [c], the contents of the root directory are listed to verify that the structure is as expected. The command `listNames()` is an extension

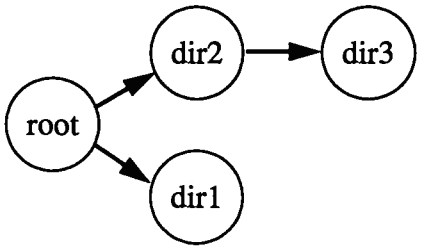

Figure 6 – Example Directory Tree

to the Raven shell. It takes a single iterable object as a parameter, iterates over it, and prints the name of each object produced. At [d], an entry for `dir2` is added to `dir1`. Although this makes the directory tree a directed acyclic graph, it does not introduce any loops and the change is permitted by RCMS. This is verified by listing the contents of `dir1`. At [e], an attempt is made to make an entry for the root in `dir1`, which would cause a loop to be created. Listing the contents of `dir1` again reveals that the entry for the root is not present – RCMS has detected the presence of the loop and rolled-back the recovery block, which in this instance consisted solely of the invocation to add the entry for root into `dir1`.

4.4.2 A Mail Transport System

The next example composite object is a very simple mail transport system. The purpose of the system is to take mail items and direct them to an underlying transport service which will deliver them to another site. Each transport service has a cost attribute, which provides some measure of the relative cost of delivering a mail item, and a status attribute which indicates whether the transport service is currently able to deliver mail items. Each mail item has an attribute which indicates the priority for delivering that item. The task of the mail transport system is twofold:

(1) Ensure that low priority mail items are not delivered via a transport mechanism which is too expensive.

(2) Ensure that if mail cannot be delivered, which may result from condition 1 and/or the inability of a transport system to accept mail items, that it is buffered until such time as delivery is possible.

The mail transport system is, of course, represented as a composite object. Figure 7 contains the interface declarations for the elementary objects used. Figure 8 contains the interface and specification clause for the composite, and Figure 9 contains the repair clause.

Interface Clause

Unlike the directory tree, the mail transport composite is encapsulated. That is, its components are not exported and, hence, its external interface consists solely of the inbound link `input`. This link receives mail items which are to be forwarded to one of the transport services. The mail transport composite has no outbound links as it is assumed that the transport services are implemented in terms of lower level primitives, not Raven inter-object links. The actual transport service objects used in this example are just stubs. They receive mail items, but rather than

```
#include <Basic.r>
#include <Array.h>

class MailItem
    {
    to, from, body: String public;
    prio : Int public;
    }

class Transport
    {
    name : String;
    up, cost : Int public;
    link in input(item:MailItem);
    constructor(p1:Int, p2:String);
    }

class DMUX
    {
    link in input(item:MailItem);
    link out lowPrio(MailItem);
    link out highPrio(MailItem);
    }

class Buffer[X]
    {
    link in input(item:X);
    link out output(X);
    behav flush();
    }
```

Figure 7 – Components of the Mail Transport System

interfacing to a transport mechanism, they simply print out some text acknowledging receipt of the item.

The mail transport composite consists of four components: one sorting object, two transport service objects, and a buffer. Unlike the directory its set of components is fixed at the time it is created. Thus, the specification doesn't contain any constraints used to define the components, it simply provides rules governing the manner in which these four components should be connected to one another.

```
#include <Basic.r>
#include <Composite.h>

composite MailTransport
    {
    link in input(item:MailItem);
    sorter : DMUX;
    t1, t2: Transport;
    hold : Buffer[MailItem];
    }

specification
    {
    define OKForHigh(X) <- X.up;
    define OKForLow(X) <- X.up & X.cost < 30;
    define AvailHigh() <- there exists X [X in {t1, t2}] OKForHigh(X);
    define AvailLow() <- there exists X [X in {t1, t2}] OKForLow(X);

    sorter = DMUX.new();
    hold = Buffer[MailItem].new();
    t1 = Transport.new(10, "Phone");
    t2 = Transport.new(40, "Internet");

    1: Connected(self, input, sorter, input);
    2: Connected(hold, output, sorter, input);
    3: there exists X Connected(sorter, lowPrio, X, input);
    4: there exists X Connected(sorter, highPrio, X, input);
    5: for all X Connected(sorter, lowPrio, X, input) -->
       X == hold | OKForLow(X);
    6: for all X Connected(sorter, highPrio, X, input) -->
       X == hold | OKForHigh(X);
    7: Connected(sorter,lowPrio, hold, input) --> ~AvailLow();
    8: Connected(sorter, highPrio, hold, input) --> ~AvailHigh();
    }
```

Figure 8 – RCMS Definition of Mail Transport Composite

Specification Clause

The component `sorter` is responsible for separating high priority mail items from low priority mail items. Mail items arrive on its link `input`. Items whose priority attribute is less than zero are output on link `lowPrio`. Items whose priority attribute is zero or greater are output on link `highPrio`. In order for mail items arriving at the composite to be forwarded to the sorting object, the forwarding link of the composite must be connected to the input link of the sorter. This is the purpose of constraint 2 in the specification clause.

```
repair
   {
   1: system.makeLink(self, input, sorter, input);
   2: system.makeLink(hold, output, sorter, input);
   3: system.makeLink(sorter, lowPrio, hold, input);
   4: system.makeLink(sorter, highPrio, hold, input);
   5: system.unLinkAll(sorter, lowPrio);
      system.makeLink(sorter, lowPrio, hold, input);
   6: system.unLinkAll(sorter, highPrio);
      system.makeLink(sorter, highPrio, hold, input);
   7: var p : IdSet[Transport] = all X : X in {t1, t2} & OKForLow(X);
      var i : Iterator[Transport] = p.getIterator();
      var t : Transport = i.next();
      system.unLinkAll(sorter, lowPrio);
      system.makeLink(sorter, lowPrio, t, input);
      hold.flush();
   8: var p : IdSet[Transport] = all X : X in {t1, t2} & OKForHigh(X);
      var i : Iterator[Transport] = p.getIterator();
      var t : Transport = i.next();
      system.unLinkAll(sorter, highPrio);
      system.makeLink(sorter, highPrio, t, input);
      hold.flush();
   }
```

Figure 9 – Repair Clause for Mail Transport Composite

The component `hold` is responsible for buffering mail items which cannot currently be delivered. Items arriving on link input are held until such time as the `flush()` behaviour is invoked, at which time all buffered items are output on link `output`. In order for these items to be re-processed, they must be fed back in to the sorting object. Consequently, link `output` of the buffer must be connected to link `input` of the sorter. This is the purpose of constraint 2.

The remaining internal linkages of the mail transport composite change over time in response to changes in the state of the transport service objects. For example, if a transport object becomes unable to accept mail items, then any links from the sorter to it must be re-directed either to another acceptable transport object or to the buffer. If the cost attribute on a transport object becomes smaller, then it is possible that low priority mail items currently being buffered can now be delivered via that transport object. The bulk of the specification clause for the mail trans-

port composite is concerned with defining what communication patterns are appropriate for different states of the transport objects. Figure 10 shows the manner in which the various

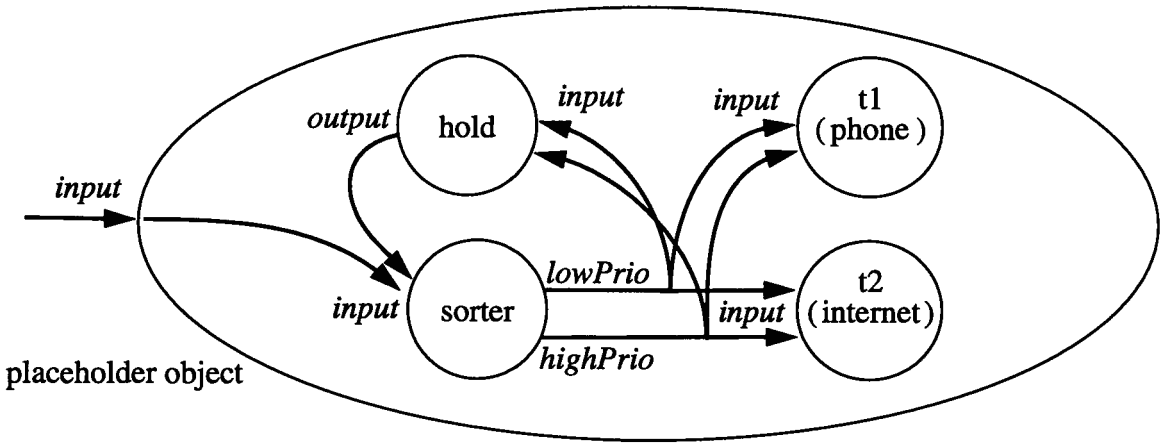

Figure 10 – Communication Structure for Mail Transport Composite

components of the composite may be interconnected. The `lowPrio` and `highPrio` outbound links of the `sorter` each have three possible targets: the buffer object `hold` and either of the two transport service objects `t1` and `t2`. Only one connection of the three possible connections may exist at a given time. The constraints of the specification clause determine which is chosen for each outbound link.

The specification clause for the composite begins with two predicates definitions, `OKForLow()` and `OKForHigh()`, which indicate when a transport object may be used for mail items of a particular priority. The only requirement for high priority mail items is that the transport object be able to accept mail items. This is indicated by its `up` attribute having the value *true*. A transport object for low priority mail must not only be able to accept mail, but its cost attribute must be less than 30 (a number chosen arbitrarily since the transport objects are simply stubs). The next two predicates, `AvailLow()` and `AvailHigh()`, simply define whether there are any transport objects which are suitable for delivering high and low priority mail respectively.

Constraints 3 and 4 state that the outbound links of the sorter must be connected to something. Constraints 5 and 6 refine this requirement. Constraint 5 states that the object to which the `lowPrio` link connects must either be the buffer object or a transport object which is acceptable for low priority mail items. Constraint 6 establishes a similar condition for the `highPrio` link. Finally, constraints 7 and 8 state that the only circumstances under which an outbound link of the sorter should be connected to the buffer object is when no acceptable transport object is available.

The only elements of the specification which have not been dealt with are the four initializing assignments which follow the predicate definitions. These assignments allocate the elementary objects which are the components of the composite and assign them to the appropriate instance variables. Because the transport objects used are just stubs, both can be instances of class Transport, differentiated only by the parameters passed when they are created. The two parameters are the initial value for the cost attribute and the a printable name which used to label the text output by the transport object.

Repair Clause

Unlike the directory tree, the mail transport composite uses roll-forward repair. Its repair clause consists of a series of repair scripts. Each repair script is preceded by an integer label as are some of the constraints. When a constraint fails, the repair script with the corresponding label is executed.

Some of the repair scripts are straightforward. For example, constraint 1 states that the output link of the buffer must be connected to the input link of the sorter. The repair script for this constraint consists of a single invocation on the system object which establishes a connection between these two links. In addition to being straightforward, this repair script, and repair

scripts 2 through 4 are only executed when the composite is first created. They simply establish the initial link connections. For reasons of simplicity, both the `lowPrio` and the `highPrio` links of the sorter are initially connected to the buffer.

Scripts 5 through 8, the core of the repair clause, consist of two pairs of structurally similar repair scripts. One member of each pair is responsible for managing the `lowPrio` link of the sorter, the other is responsible for the `highPrio` link. Script 5 executes when the `OKFor-Low()` predicate of constraint 5 fails, indicating that the transport service to which the `lowP-rio` link is connected is no longer suitable for low priority mail items. The first invocation of script 5 removes the connection between `lowPrio` and the transport object. The second object establishes a connection between `lowPrio` and the buffer.

Script 7 executes when the `lowPrio` link is connected to the buffer, but a suitable transport service for low priority mail items is available. The first line of the script is a variable declaration which uses a set constructor as an initializing expression. This is an example in which the basic syntax of Raven has been augmented with RCMS expressions. Raven allows initialization of variables, but only within RCMS scripts may the initialization expression be a set constructor. The set constructed consists of all the transport service objects which are suitable for delivering low priority mail. The next two lines of the script effectively extract a single transport service at random from the set. The `lowPrio` link is disconnected from the buffer and connected to the chosen transport object. Finally, the `flush()` behaviour of the buffer is invoked, causing any buffered mail items to be removed from the buffer and re-sent to the sorter.

It should be intuitively clear that the execution of these scripts in response the failure of the corresponding constraints leads to an acceptable configuration of the mail transport composite. However, because roll-forward repair is used, the question arises as to when the composite should be validated and, if necessary, repaired. Use of roll-back repair in the directory tree pro-

vided a natural trigger for validation: exit from a top-level recovery block. With roll-forward repair there are no system activities with which validation/repair may be integrated. What is required is a system to monitor the composite and execute the validation/repair phase.

There are a variety of possible monitoring strategies. Currently, RCMS supports three styles of composite monitoring when roll-forward repair is used.[1] The first, called *real time*, causes RCMS to intercept all invocations on components, permitting it to detect changes in their instance variables. When such a change occurs, RCMS re-validates and, if necessary, repairs the composite. The second, called *polling*, involves executing the re-validation/repair phase at regular intervals. The third, called *error trapping*, performs re-validation/repair only on the occurrence of certain failures detected by the system. Currently the only failure which is trapped by RCMS is failure of an invocation on an object at a remote site. Such a failure could occur as the result of a network or site failure, or the termination of an object due to a programming error.

The monitoring mechanism used is not specified within the composite definition. Rather, there is a special RCMS manager object on each node which supports behaviours for associating monitoring methods with a composite. Any combination of the three types of monitoring is legal for a composite which uses roll-forward repair. For the mail transport composite, polling is used. Figure 11 shows an interactive session using the mail transport composite. The RCMS manager object is accessed via the system object. The mail transport composite is registered for monitoring by periodic polling using the `pollComposite()` behaviour. This behaviour takes two parameters: the first is a capability for the composite to be monitored, and the second is an integer indicating how frequently the polling should be done. In the current implementation this value is specified in units of one hundredth of a second.

1. As noted in Section 4.2.6, when roll-back repair is used, the monitoring mechanism is combined with the repair mechanism and composite verification occurs upon exiting from a top-level recovery block.

Because the mail transport composite is accessed both by the RCMS manager object, and by other objects for the purpose of delivering mail items, the issue of concurrency control arises. Raven provides support for object locking, but not for transactions. RCMS is able to make use of Raven's locking support to prevent some types of unwanted concurrent access to the components of a composite object. However, complete avoidance of inconsistencies caused by concurrent access awaits the implementation of transaction services in Raven. The RCMS model assumes that components objects are locked during the validation/repair phase. Several more interesting aspects of the locking issues are discussed in Section 6.3.

Returning again to Figure 11, we can see the mail transport composite in action. As with all composites, an initial round of validation/repair is performed when the composite is created. Thus, the composite is immediately ready for use. At [a], a simple driver routine called `sendMail()` is invoked. Its parameters are the name of the sender, the name of the receiver and the priority of the mail item. The driver routine creates a mail item with those values, establishes a connection to the mail transport composite's inbound link, and sends the mail item. The response from the system at [b] indicates that the mail item has been forwarded to the internet transport service. The next two lines show a similar action and response for a mail item with low priority. Note that it is forwarded to the phone transport service because the internet transport

```
$ m = MailTransport.new()
$ system.RCMSManager.pollComposite(m, 10)
$ sendMail("Terry", "Don", 10)                          [a]
Internet: Received item (Terry --> Don)                 [b]
$ sendMail("Terry", "Don", -10)
Phone: Received item (Terry --> Don)
$ m.t1.up = 0                                           [c]
$ sendMail("Terry", "Don", -10)                         [d]
$ sendMail("Terry", "Don", 10)                          [e]
Internet: Received item (Terry --> Don)
$ m.t1.up = 1                                           [f]
Phone: Received item (Terry --> Don)
```

Figure 11 – Interactive Session with Mail Transport Composite

service is too expensive for low priority mail items. At [c], the failure of the phone transport service is simulated by setting its up attribute to 0. At [d], another low priority mail item is sent. There is no response from either transport object because the mail item has been directed to the buffer. With the phone transport service down, there is no acceptable transport service available for low priority messages. At [e], a high priority message is sent, and is handled just as before. At [f], renewed availability of the phone transport service is simulated by setting its up attribute to 1. The connection from the sorter to the phone transport service is re-established, the buffer is flushed, and the buffered mail item arrives safely at the phone transport service for delivery. Thus, the mail transport service responds to changes in the environment as was desired.

Design Alternatives for the Mail Transport Composite

However, the use of polling as the monitoring mechanism for this example reflects the needs of exposition more than of good design. While it has been convenient to present the polling capabilities of RCMS in the context of the mail transport example, it also permits invalid configurations to exist transiently. Consider the situation in which the phone transport service goes down, followed very closely by the arrival of a low priority message at the input of the composite. If the composite is not polled by RCMS in between these two events, then the message is forwarded to the phone transport service despite the fact that the underlying transport service is not available. Given that it cannot deliver the message, the phone transport object can either buffer it or drop it. If the phone transport object were to drop the message then the mail transport composite would be unreliable. At first this might seem entirely unacceptable, but, in fact, it forces us to examine the overall design of the system.

There are a number of ways in which the mail transport system could have been constructed. Determining which design is appropriate is a matter of examining the design trade-offs associated with each. As the purpose of this chapter is to introduce the features of RCMS, a full-scale

analysis of the problem is not appropriate. However, it is worthwhile to consider some of the alternatives in order to gain perspective on how the features of the configuration system affect the design process. Consider three possible designs for the mail transport system:

(1) The transport objects drop messages which arrive when the underlying transport service is unavailable. Invocation interception is used to monitor the composite, allowing RCMS to be informed of changes in the transport objects as soon as they occur.

(2) The transport objects maintain a small number of buffers for messages which arrive when the underlying transport service is unavailable. Polling is used to monitor the composite. The buffering ensures that no messages are lost during the short periods when the composite is not valid.

(3) The transport objects drop messages which arrive when the underlying transport service is unavailable. Polling is used to monitor the composite. The software using the mail transport composite is modified to perform end-to-end checking, and it re-transmits lost messages.

The question of which design is most appropriate can be answered only by considering the costs and benefits associated with each. However, because the mail transport composite is only an example, not part of a working system, many of the costs are unknown; an authoritative answer is not possible. The type of information which would guide the choice of a design in this situation is:

(1) the relative cost of polling and invocation interception as the monitoring mechanism,

(2) the difficulty in modifying the transport object software to incorporate buffering,

(3) whether end-to-end checking is already a requirement for overall mail system reliability.

These different designs and the factors affecting the choice of a design illustrate that a configuration management tool cannot be considered in isolation. It is part of collection of resources which a programmer uses to achieve a goal. Creating a system which provides a particular service involves trade-offs. The choice of monitoring mechanisms in RCMS, each with different characteristics, provides flexibility in responding to design constraints imposed from outside of RCMS. Further work with RCMS may suggest even more possibilities of this sort.

4.4.3 Automatic Re-compilation System

Part of the original motivation for creating RCMS was desire to have a general configuration management tool, that is, one which could handle a variety of tasks: maintaining correctness, as in the example of preventing cycles in a directory tree, on-the-fly restructuring of a system, as in the mail transport example, and assisting in providing a software engineering environment, which brings us to the current example, the third of the chapter. The composite object described in this section provides a very simple facility for keeping programs up to date in much the same way that `make` does. The composite makes use of a number of elementary classes representing various sorts of files. The definitions of these classes are given in Figure 12. These objects are, in fact, only stubs. The only real action of behaviours such as `edit()` and `compile()` is to modify the date of last modification of the objects involved. The `CFile` object is the most interesting of the group because it maintains several pieces of information which are central to the specification of the composite. Instance variable `imports` of `CFile` is a set containing references to the header files which that `CFile` uses. This set could be maintained by the `CFile` object by parsing its contents, but in this example the references are maintained manually. Instance variable `produces` of `CFile` contains a reference to the object file which results from compiling the C file. Behaviour `compile()` of `CFile` causes the object file referred to by `produces` to be updated. Finally, behaviour `linkFiles()` of `ExecFile` takes a set of C files as arguments and uses the object files associated with each to update itself.

```
#include <Basic.r>
#include <IdSet.h>

class HFile
    {
    lastModDate : Int public;
    behav edit();
    }

class OFile
    {
    lastModDate : Int public;
    }

class CFile
    {
    lastModDate : Int public;
    imports : IdSet[HFile] public;
    produces : OFile public;
    behav edit();
    behav compile();
    }

class ExecFile
    {
    lastModDate : Int public;
    behav linkFiles(modules : IdSet[CFile]);
    }
```

Figure 12 – Components of Automatic Re-compilation System

Interface Clause

The purpose of the composite, which is called `Application`, is to keep the executable for a single application up-to-date with respect to the files it is compiled from. The complete definition of `Application` is given in Figure 13. The composite has just two instance variables, one, called `source`, containing the set of C files out of which the application is constructed and a second, called `exec`, containing the object representing the executable for the application. There is no need for the object files resulting from compilation, nor for the header files, to be represented in this simple example because they can be referred to via the C file objects. In a more complete implementation of the features of make, these two sets of objects might be

```
#include <Basic.r>
#include <Composite.h>
#include "Components.h"

composite Application
    {
    source : IdSet[CFile] public;
    exec : ExecFile public;
    }

specification
    {
    source = IdSet[CFile].new();
    exec = ExecFile.new();
    1: for all CFILE [CFILE in source]
       CFILE.produces.lastModDate >= CFILE.lastModDate;
    2: for all CFILE [CFILE in source]
        for all HFILE [HFILE in CFILE.imports]
          CFILE.produces.lastModDate >= HFILE.lastModDate;
    3: for all CFILE [CFILE in source]
        exec.lastModDate >= CFILE.produces.lastModDate;
    }

repair
    {
    1: var old : IdSet[CFile] = all CFILE : CFILE in source &
          CFILE.lastModDate > CFILE.produces.lastModDate;
       var i : Iterator[CFile] = old.getIterator();
       var aCFile : CFile = i.next();
       aCFile.compile();
    2: var old : IdSet[CFile] = all CFILE : CFILE in source &
          there exists HFILE [HFILE in CFILE.imports]
          HFILE.lastModDate > CFILE.produces.lastModDate;
       var i : Iterator[CFile] = old.getIterator();
       var aCFile : CFile = i.next();
       aCFile.compile();
    3: exec.linkFiles(source);
    }
```

Figure 13 – RCMS Definition of Automatic Re-compilation Composite

explicitly represented. Furthermore, since both sets of objects can be expressed in terms of their relationships to the C files, the contents of these two sets could be maintained automatically by RCMS by using dynamic set constructors and temporal assignment.

Specification and Repair Clauses

The constraints required to ensure that the executable for the application is up-to-date are straightforward. Constraint 1 states that the last modification date of the object file produced from a given C file must be greater than the last modification date of the C file. In other words, if the C file is altered, it must be re-compiled. The repair script for constraint 1 does just this. Constraint 2 establishes a similar condition with respect to any of the header files which a given C file imports. Once again, the repair script re-compiles the necessary C file. The final constraint states that the last modification date of the executable must be greater than the last modification dates of all the object files from which it is generated. The corresponding repair script re-links the executable.

Figure 14 shows an interactive session using this composite. Some of the initialization is not shown. For example, the C file and header files objects had to be created and the imports fields of the C files properly initialized. Also, it is assumed that when the C file are created that their corresponding object files are up to date. In this example, there are three C files, called `Chord`, `Iso`, and `Display` (the application is one which formats songs with guitar chord notation). C file `Chord` is the main program, and its imports the header files for `Iso` and `Display`, which it uses. These two header files are `IsoDef` and `DisplayDef`, respectively. In addition, `Iso` and `Display` import their own header files.

The session of Figure 14 begins with the allocation of a new `Application` composite. Next, the C files of the application are added to instance variable `source`. This provides the composite with enough information to carry out its task. Thus, we invoked the `edit()` behaviour on the C file `Chord`. In this example, we need to have precise control over when the validation/repair procedure occurs. For example, if we change part of the interface exported by the C file `Display`, then changes have to be made to both the C file itself and to its header file. It is unde-

```
$ app = Application.new()
$ app.source.add(Chord)
$ app.source.add(Iso)
$ app.source.add(Display)
$ Chord.edit()
$ app.checkAndRepair()
*** Compiling: Chord
*** Re-linking (3 object files)
$ IsoDef.edit()
$ app.checkAndRepair()
*** Compiling Chord
*** Compiling Iso
*** Re-linking (3 object files)
```

Figure 14 – Interactive Session with Automatic Re-compilation Composite

sirable for RCMS to attempt re-compilation at a point in time where these files are inconsistent. Therefore, automatic validation/repair when any of the objects change state is not appropriate. In this example, validation/repair is initiated manually by invoking the `checkAndRepair()` behaviour of the composite.[1] The validation/repair process re-compiles C file `Chord` and then re-links the executable; precisely the set of actions expected given that `Chord` was edited. Next we edit the header file `IsoDef`. Validation/repair is manually initiated again. This time both `Chord` and `Iso` are re-compiled, since both import that header file, then the executable is re-linked. Thus, the composite provides a very simple `make`-like service.

4.4.4 A Distributed Composite

The final example of this chapter is more of a pedagogical example, intended to demonstrate the use of RCMS with a composite which is distributed across two systems. The monitoring mechanism used in this case is detection of a remote invocation failure, a failure which results in two of the components of the composite being isolated from one another.

1. If Raven supported a user level transaction-like mechanism for grouping operations together it might be possible to automatically initiate validation/repair at the end of a transaction, but this is not supported in the current implementation.

The composite object for this example consists of three components: two filters and a buffer. The two filter objects reside on different machines. Under normal circumstances, the input of the composite is forwarded to the input of the first filter, the output of the first filter is sent to the input of the second, and finally, the output of the second filter is forwarded to the output of the composite. Externally, the composite appears to be a simple filter with an input and an output. The buffer resides on the same machine as the first filter. Should communication with the second filter fail, output from the first filter is routed to the buffer, where it is held until such time as

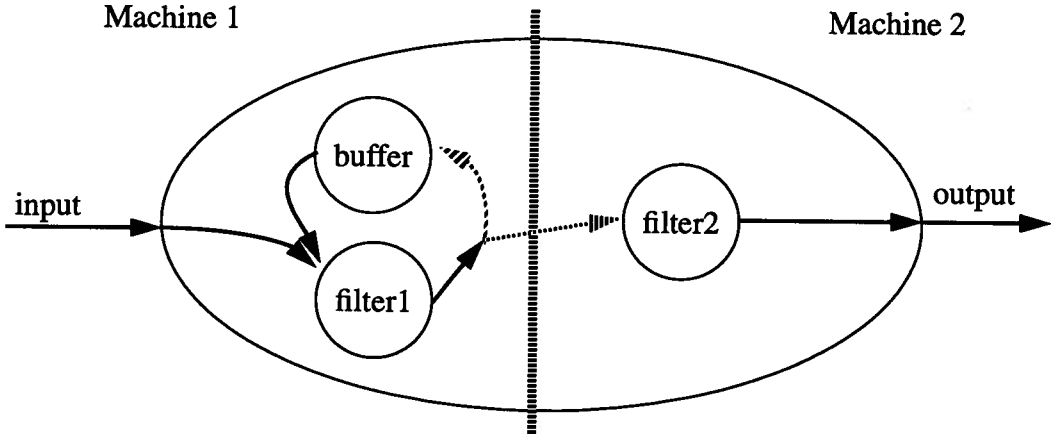

Figure 15 – Structure of Distributed Filter Composite

communication can be re-established with the second filter. The assumption is, of course, that the second filter has not actually failed, merely that communication with it has been interrupted. Figure 15 shows the structure of the distributed filter composite.

Figure 16 provides the interface definitions for the components of this composite. The two filters used are instances of the same Raven class. Each filter object is parameterized at the time of its creation by a single string. Values received on its inbound link are compared to this string, and if they differ the received value is sent on its outbound link. The buffer object is an instance of the same class used in the mail transport example.

```
#include <Basic.r>
#include <Array.h>

class Filter
    {
    link in input(p1:String);
    link out output(String);
    constructor(p1:String);
    }

class Buffer[X]
    {
    link in input(item:X);
    link out output(String);
    behav flush();
    }
```

Figure 16 – Interface Definitions for Components of Distributed Filter

Figure 17 provides the source code for the distributed filter composite. Only a few new features of RCMS are used in this example, so for the most part it looks similar to the mail transport composite. The composite is parameterized at creation time by the two filter objects. This allows the two filter objects to be created on separate machines and then passed in when the composite is created. The significant element of the specification clause is the use of the built-in predicate `Connectable()`, which evaluates to *true* if communication along the specified link is possible.

Constraint 3 states that if communication with `filter2` is possible, then the output of `filter1` should be connected to the input of `filter2`, and the output of `filter2` should be forwarded to the output of the composite. A more natural form for the specification might have a separate constraint which simply indicated that the output of `filter2` should be forwarded the output of the composite. After all, this connection should not really change. The motivation for writing constraint 3 in the manner shown is concern over the location at which validation, repair, and forwarding occur.

```
#include <Basic.r>
#include <Composite.h>

composite DistFilter
    {
    filter1, filter2 : Filter;
    buffer : Buffer[String];
    link in input(p1:String);
    link out output(String);
    constructor(f1, f2: Filter);
    }

specification
    {
    filter1 = f1;
    filter2 = f2;
    buffer = Buffer[String].new();

    1: Connected(self, input, filter1, input);
    2: Connected(buffer, output, filter1, input);
    3: Connectable(filter1, output, filter2, input) -->
           Connected(filter1, output, filter2, input) &
           Connected(filter2, output, self, output);
    4: ~Connectable(filter1, output, filter2, input) -->
           Connected(filter1, output, buffer, input);
    }

repair
    {
    1: system.makeLink(self, input, filter1, input);
    2: system.makeLink(buffer, output, filter1, input);
    3: system.unLinkAll(filter1, output);
       system.makeLink(filter1, output, filter2, input);
       system.makeLink(filter2, output, self, output);
       buffer.flush();
    4: system.unLinkAll(filter1, output);
       system.makeLink(filter1, output, buffer, input);
    }
```

Figure 17 – Definition of the Distributed Filter Composite

Limitations of Fault Tolerance in RCMS

In the current implementation of RCMS, certain functions are localized to a single site.[1] In particular, execution of the code which performs validation and repair occurs on the machine on

1. The reasons for this are made apparent in Chapter 5, while a discussion of improved distribution of these functions occurs in Chapter 6.

which a composite was created. In this example the assumption is that the composite and `filter1` reside on one machine, hereafter referred to as machine 1, while `filter2` resides on a different machine, hereafter referred to as machine 2. Suppose that the constraint linking the output of `filter2` to the composite were extracted from constraint 3 and placed in a constraint of its own. The validation code responsible for checking this constraint executes on machine 1. However, in order to verify that a link exists, the system requires access to both machine 1 and machine 2. If machine 2 becomes inaccessible due to communications difficulties, then the constraint linking `filter2` to the composite fails because the system is unable to verify that the link exists. Furthermore, the constraint cannot be satisfied as long as machine 2 is inaccessible. Constraint 3 avoids this problem by ensuring that the link between the filter and the composite is only checked for when machine 2 is accessible.

The ability of this composite to respond to failures is limited. The only type of failure addressed by this design is a transient loss of connectivity between `filter1` and `filter2`. RCMS supports a monitoring style suited to this sort of composite. If data is output to a link and a communication failure occurs, the Raven run-time system signals an error. RCMS traps this error, and if the object sending the data is part of a composite, it performs validation and repair for that composite. The `monitorFailure()` behaviour of the RCMS manager object is used to register a composite for this form of monitoring.

Figure 18 shows one of two sessions involved in demonstrating the distributed filter. Another session running on a different machine was required for the creation of the second filter component. At [a], the Raven system behaviour `getRemoteCapability()` is used to obtain a capability for the second filter object. Variables `host` and `local` contain values which identify the remote machine on which the filter object resides.[1] Having created one filter locally, and

1. The variable `host` contains the internet address for the remote machine and the variable `port` contains the UDP port number of the Raven process on that machine which contains the other filter object.

```
$ f1 = Filter.new("world")
$ f2 = Filter.getRemoteCapability(host, local)                [a]
$ distFilt = Test.new(f1, f2);                                [b]
$ system.RCMSManager.monitorFailure(distFilt)
$ sendItem("one")                                             [c]
** Received one **                                           [d]
$ sendItem("world")
...                                                           [e]
$ sendItem("two")                                             [f]
$ sendItem("hello")
...                                                           [g]
** Received two **                                           [h]
```

Figure 18 – Interactive Session with Distributed Filter

obtained a capability for the one on the remote machine, the composite itself is created at [b]. At [c], a value is sent to the composite. The two filters inside the composite filter out the strings "hello" and "world." As the string sent in this case is "one" it passes through the filter and arrives at an object which prints the message seen at [d]. At [e], some manipulation is performed on the remote machine to make the second filter inaccessible. At [f] and the line following, values are sent to the composite. At [g], some further manipulation on the remote machine makes the second filter accessible again, and, as a result, at [h], the buffered value "two" makes it through the filter. The buffered value "hello" is not displayed since it is removed by the second filter.

The distributed filter composite points up shortcomings of both RCMS and the Raven system. As noted in the preceding section the distributed filter possesses a limited amount of fault tolerance. The most significant problem with the current implementation of RCMS is that a composite object represents a single point of failure. If the machine on which a composite object crashes then that composite object effectively ceases to exist, despite the fact that many, or even all, of its components continue to exist on other machines. Another problem is that RCMS needs to address the issue that link failure arises from two different underlying problems: (1) communi-

cations failure making an object inaccessible, (2) termination of an object resulting from a software or hardware failure. In the distributed filter composite if the second filter object is terminated, then the composite simply buffers items ad infinitum rather than generating a replacement for the missing filter. Of course, the specification could have been written such that a new filter was created. Unfortunately, the underlying problem is not solved. Imagine that the composite is constructed, not of simple filter objects, but rather, of more complex objects which do not permit duplicate objects to exist. Imagine further that the specification for the composite has been written such that a new object is allocated when communication with the existing one is not possible (as suggested above for the distributed filter composite). Now a communications failure occurs, and causes RCMS to allocate a new object despite the fact that the old one still exists, it is simply unreachable for the moment. When the communications failure is repaired, the duplicate objects become aware of each other and the proscription against duplicate objects is violated. Thus, the inability to distinguish between communications failures and the failure of objects at remote sites represents a significant area of further research for RCMS.

Investigating the fault tolerance of the distributed filter also revealed the importance to configuration management of a flexible exception handling mechanism in Raven. Imagine that a message is sent from `filter1` to `filter2`, following which the communications link between machine 1 and machine 2 fails. Since it has a message, `filter2` attempts to send it on its output link. However, since machine 1 cannot be reached this send fails, and the message is lost. If `filter2` were capable of recognizing and responding to this failure it might attempt to re-send the message at some time in the future (assuming that communications would be restored). However, the current exception handling facilities in Raven are primitive, and it would be quite complex to implement a filter object capable of operating in this manner. Hence, the lack of an easy to use exception handling facility in Raven reduces the fault tolerance of the distributed fil-

ter composite. This is further evidence of the interaction between the design of a configuration management system and the design of the system of which it is a part.

4.5 Summary

In this chapter we have seen how the model of configuration management developed in Chapter 2 has been transformed into a working configuration management system. The result is a configuration management system which is general, active, and distributed. The key element of RCMS is the composite object, a concrete embodiment of a configuration. The composite object contains all the configuration information for a particular configuration: identification of components, specification of behaviour, and specification of repair. The composite object is, in fact, a configuration management agent.

In the latter part of the chapter, we have seen how RCMS is used to define a variety of composite objects which cover a range of configuration management activities. Some of these composite objects reveal interesting features about the relationship of the model to the implementation. The mail transport composite, for example, demonstrates an interesting interaction between the design of a composite object and the actual mechanism which is used for monitoring that composite.

The model uses check sets to describe how configurations are monitored. It does not specify, however, precisely what events should be part of the check set. One of the most interesting aspects of developing RCMS has been the design and implementation of monitoring techniques. The most powerful monitoring technique (real time), also imposes the highest overhead. While less powerful techniques may be used to improve efficiency, they affect the design of the composite objects and may impose other limitations. Given the trade-offs encountered in designing the mail transport composite, it appears that a configuration management system needs to pro-

vide the user with a variety of monitoring techniques. In addition, the importance of monitoring led to further investigation of techniques to improve its efficiency, which is documented in Section 6.2.

A second interesting issue developed in the process of implementing the distributed filter composite. As discussed in Section 4.4.4, the degree of fault tolerance offered by the current version of RCMS is limited. The limitation stems, in part, from a lack of redundancy in composite objects. This limitation motivated exploration of mechanisms to enhance redundancy in RCMS which are discussed in Section 6.4. Other limitations in fault tolerance stem from inadequate error handling facilities in the Raven system. The distributed filter composite highlights how the limited error handling facilities available to the Raven programmer affect the degree of fault tolerance which can be offered by RCMS. Overall, improved fault tolerance remains an area in which there is much work to be done in RCMS.

CHAPTER 5

Implementation of RCMS

This chapter provides a detailed description of the various pieces of software which together constitute the Raven Configuration Management System. The description is confined to the basic elements of the system. Discussion of the advanced features and future work is presented in Chapter 6. RCMS consists of:

(1) a compiler which translates RCMS composite definition into Raven classes,

(2) a class library containing Raven classes used by RCMS,

(3) run-time routines which assist in object monitoring and repair.

We examine each of these items in turn.

5.1 The RCMS Compiler

A programmer wanting to create composite objects uses a tool called `rcfg`. Currently, `rcfg` runs under Unix and produces Unix executables. When these executables are run a Raven vir-

tual machine is created, and, in the context of this virtual machine the programmer can create objects and invoke behaviours on them. A Raven shell can be included as part of the virtual machine, allowing the programmer to interact in real time with the Raven system. All of the examples shown in Chapter 4 were executed using the Raven shell.

The `rcfg` program is actually a very simple driver which executes a number of other programs in order to carry out its work. RCMS source files are translated into Raven source files via a program called `RvCfgTrans`. Raven files are translated into C files via `RavenToC`, and finally `gcc` is used to compile and link all C source files into an executable. The link process automatically includes several libraries which provide services such as lightweight threads, Raven run-time support, RCMS run-time support, and the basic Raven/RCMS class libraries.

At the level of the Raven virtual machine, there is no distinction between an elementary object and a composite object. The basic structure of the Raven system was not changed in order to support RCMS. Consequently, as the Raven system evolves, RCMS takes advantage of that evolution with only minimal changes to the RCMS support services. Because of this, each RCMS composite class is represented in the Raven virtual machine by a corresponding elementary class, known as the composite's *placeholder* class. The task of the RCMS compiler is to read an RCMS source file and produce a Raven source file containing placeholder class definitions and any additional code required to make those definitions acceptable to the Raven compiler.

5.1.1 Compiler Internals

An RCMS source file is composed of three elements: Raven class interface declarations, RCMS composite interface declarations, and RCMS composite definitions (which consist of an interface and specification/repair clauses). Imported declarations, either for Raven classes or for other composites, are supported by textual inclusion of header files. Thus, imported declarations

are indistinguishable from local declarations. Typical input to the RCMS compiler, thus, consists of a series of Raven class interface declarations resulting from the inclusion of standard header files, followed by the RCMS composite definitions written by the programmer.

The RCMS compiler works in two phases. During the first phase, the RCMS source file is parsed and a table of class descriptions is created. In the second pass, this table is scanned and a series of Raven class declarations and definitions are emitted. Declarations for elementary classes are emitted with little processing since their form in RCMS and Raven are identical. The only complication arises from the need to emit the definitions in an order which is acceptable to the Raven compiler. Raven is a "define before use" language, meaning that every class must be defined before it can be referred to. Forward declarations are permitted. Class references are created when defining the types of instance variables, parameters, return values and inheritance. For example, a simple stack class:

```
class Stack <- Object
    {
    contents : Array[Int];
    top : Int;
    behav push(value : Int);
    behav pop() : Int;
    }
```

makes reference to class `Object`, class `Array`, and class `Int`. Thus, these classes must have been defined before the compiler encounters the above code.

5.1.2 Code Generation for Elementary Classes

Elementary class definitions are output first. The global class table is scanned and elementary class declarations emitted. Upon locating a suitable elementary class declaration, the compiler must first ensure that any classes referred to by it have already been emitted. A set of mutually recursive routines is used to achieve this. Effectively these routines construct a graph whose nodes are class declarations and whose edges represent references from one class to another.

Cycles in the graph represent declarations with mutual references which require a forward declaration to be used for one of the declarations in the cycle. Starting at the node representing the class declaration to be emitted, a post-order traversal of the graph is performed. A hash table is used to record node markings which permits the detection of cycles. When a cycle is detected a forward declaration for that node is emitted and it is treated as a leaf with respect to the traversal operation. When the traversal returns to a given node after traversing all of its descendents, a full declaration is emitted, and it is marked in the table as complete. Eventually, the traversal returns to the original node, representing the declaration which the compiler initially decided to emit. The traversal process guarantees that any classes which it refers to have now been themselves emitted, hence, the declaration for that class itself can be emitted. Scanning of the class table resumes, skipping those which are marked as complete.

5.1.3 Code Generation for Composite Classes

Composite class definitions require substantially more processing than elementary class interface declarations. The interface, specification, and repair clauses must be translated to an elementary class definition and accompanying behaviour definitions. The interface of the placeholder class is a combination of the interface of the composite class, a common interface shared by all placeholder classes, and an interface derived from the contents of the specification and repair clauses. The behaviours of the placeholder class perform functions such as validation of the composite, computing the current components, and evaluation of predicates.

Recall that the interface of a composite consists of instance variable declarations and link declarations. Each instance variable declaration of the composite becomes an instance variable declaration of the placeholder. The keyword `public` is carried through to the placeholder declaration. The keyword `indirect` has no effect on the form of the instance variable declara-

tion used in the placeholder, but does cause additional code to be emitted in the behaviour which computes the components of the composite.

An inbound link declaration is translated into both inbound and outbound link definitions of the same name. The inbound link declaration is coupled with an implementation which simply takes the data arriving on the link and sends it out on the outbound link. Similarly, an outbound link becomes both an inbound and an outbound link, with the implementation of the inbound link forwarding data to the outbound link. This doubling up of link declarations combined with implementations for the inbound links which simply forward data, provides the desired behaviour of links on composite objects. Figure 19 shows the source code for a simple composite with

```
composite Simple <- Composite
    {
    f1 : SimpleFilter;
    link in input(p1:Int);
    link out(Int);
    }
```

Figure 19 – Simple Composite with Links

an inbound and an outbound link. Figure 20 shows those portions of the output of the compiler relevant to the link declarations. The placeholder object acts as a gateway into the composite. An external connection to `input` becomes a connection to the inbound link `input` of the placeholder object. A connection from the input of the filter component to `input` becomes a connection to the outbound link `input`. When data arrives at the inbound link `input`, the implementation forwards the data to the outbound link `input`, and so it arrives at the filter component. The output link mechanism is similar.

```
class Simple
    {
    f1 : Filter;
    link in input(p1:Int);
    link out input(Int);
    link out output(Int);
    link in output(p1:Int);
    }

link input
    {
    send input(p1);
    }

link output
    {
    send output(p1);
    }
```

Figure 20 – Implementation of Inter-Object Links

Computing the Components of Composite

The remaining portions of the placeholder definition are derived more indirectly from the composite definition. The first item that we will consider is a behaviour called `updateComponents()` which computes the components of the composite. The capabilities for all components are stored in an instance variable named `components` of type `IdSet[Object]`.

The components are the contents of indirect instance variables together with the non-indirect instance variables. At first glance, the task of `updateComponents()` would seem straightforward: add non-indirect instance variables to `components`, then add the contents of indirect instance variables. However, for each instance variable which is subject to a temporal assignment constraint (the `:=` operator), `updateComponents()` must first ensure that the expression on the right hand side of the assignment is computed and assigned to the instance variable. Compilation of logical expressions is required at several different points in the compiler, and

there is a common set of routines used for this purpose which are described later in this section. These routines are used to compile the right hand side of the assignment operator into a segment of Raven code which computes its value. This value is assigned to the appropriate instance variable. All instance variables which are the targets of temporal assignments are updated in this fashion prior to computing the components set.

The behaviour `updateComponents()` has one other responsibility. Normal operation of RCMS requires a function which takes a given object and returns the set of composites of which that object is a component. For example, when an object changes state RCMS must be able to determine which composite objects it belongs to in order to know that those composites require validation/repair. This is implemented by storing within each object a set of references to the composites to which it belongs. Because the components of a composite vary during its existence, it is necessary to update these references.

Upon entering the `updateComponents()` behaviour, the current value of `components` is assigned to a local variable called `oldComponents`. After a new value for `components` has been obtained, the difference between `components` and `oldComponents` is computed. This represents the set of objects which were previously components, but no longer are. This difference set is iterated over and the references from the objects back to the composite are removed. The difference between `oldComponents` and `components` is then computed. This represents the set of objects newly added to the composite. This difference set is also iterated over, and references from each object to the composite are added.

Evaluating the Specification of a Composite

The behaviour `verify()` is responsible for determining whether a composite object is valid. For each constraint of the specification clause of the composite definition, not including initial

and temporal assignments, the `verify()` behaviour has a segment of code which evaluates the constraint. Since these constraints are simply logical expressions, the common expression compilation subroutines are used. If any constraint evaluates to *false*, `verify()` returns *false*, otherwise it returns *true*. Prior to evaluating each constraint, the instance variable `currentConstraint` is set to the value of the integer label of that constraint, if it has one, and -1 otherwise. Thus, if `verify()` returns *false*, `currentConstraint` indicates which constraint caused the failure.

Initial and temporal assignments are handled separately. As noted in the discussion of `updateComponents()`, temporal assignments cause the generation of code which re-evaluates the right hand side of the assignment and stores the resulting value to the instance variable indicated by the left hand side. Thus, instance variables for which temporal assignment constraints exist are given new values whenever `updateComponents()` is invoked. Initial assignments are handled in a similar manner, except that evaluation of the right hand side occurs in the `constructor()` for the composite object. The `constructor()` is executed once, when the composite is created. The `constructor()` also contains an invocation to `updateComponents()` which ensures that the components set is correct for the newly created object.

Performing Repair for a Composite

The behaviour `repairComponents()` carries out repair activities for non-recoverable composite objects. Repair for recoverable composites is integrated with Raven's recovery manager. Each repair script of a composite is compiled into a private behaviour. Hence, `repairComponents()` simply switches on the value of `currentConstraint` and invokes the corresponding repair behaviour. The validation/repair phase thus consists of a loop which repeatedly calls `updateComponents()`, `verify()`, and `repairComponents()` until such time as `verify()` returns *true*.

Compilation of the repair scripts is very straightforward. Since the syntax for the scripts is essentially that of Raven and the target language of the compiler is also Raven, most of the compilation process simply involves passing the statements of the script through to the output. The only complication is that RCMS allows the use of its logical expression language for initialization of variables. This is handled by using the expression compilation subroutines to generate the Raven code for the expressions.

Code Generation for Predicates

The RCMS compiler also generates one private behaviour for each predicate defined in the specification clause of the composite. The parameters passed into such a behaviour are simply the arguments declared for the predicate. Because RCMS predicates are not currently type-checked, the parameters are declared with type `cap`. Invocations on variables of type `cap` are not type-checked by the Raven compiler. Hence, the lack of type checking in RCMS is propagated through to the generated Raven code. The body of a predicate is simply a logical expression, which means that the common expression code generation routines are used to create the body of the behaviour which represents the predicate. Figure 21 contains the code generated for the predicate:

```
define OKForLow(X) <- X.up & X.cost < 30.
```

which was used in the mail transport composite of Section 4.4.2. As can be seen the code is quite straightforward. The body of the predicate translates to a simple Raven expression which directly accesses the `up` and `cost` instance variable of the parameter `X`, which is assumed to be a transport service object. The only mysterious elements of this behaviour definition are the invocation of `checkActiveAndAdd()` and `active.removeLast()`.

These two invocations serve to detect recursive predicate evaluations which would otherwise lead to infinite looping during the evaluation of some expressions. Their purpose is similar to

```
behaviour OKForLow
    {
    var tmp : Int;
    if (checkActiveAndAdd("OKForLow", 1, X)) return False;
    tmp = ((X).up && ((X).cost < 30));
    active.removeLast();
    return tmp;
    }
```

Figure 21 – Code Generated for a Simple Predicate

that of memoing functions used in functional programming languages [23][27]. In functional programming languages, function evaluation has no side-effects. Consequently, two evaluations of a given function with the same arguments necessarily produce the same result. A memoing function is one which caches previously computed values, so that if an evaluation of a function with a particular set of arguments has already been performed, then the previously computed answer may be returned. The purpose in using memoing functions is increased efficiency. Assuming a good cache implementation, looking up a cached value and returning can be considerably more efficient than re-computing that value.

RCMS uses a variant of memoing functions, not to improve performance, but to provide correct behaviour for recursively defined predicates. If RCMS did not use memoing functions, then the evaluation of some recursively defined predicates on cyclic data structures would result in non-terminating predicate evaluations. Consider the `Path()` predicate used in the directory tree example of Section 4.4.1 and the cyclic directory structure shown in Figure 22. An evaluation of `Path(root, dir3)` begins with a check to see whether `dir3` is directly contained in `root`. This is not true and consequently existential quantification is used to iterate over the directories contained in `root`, checking to see if there is a path from one of them to `dir3`. Assuming that the iteration begins with `dir1`, `Path(dir1, dir3)` is recursively evaluated. Once again, `dir3` is not directly contained within `dir1`, and so its contents are iterated over. The only entry

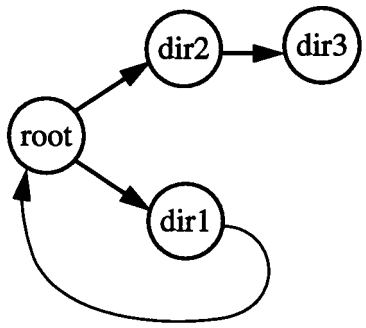

Figure 22 – Cyclic Directory Structure

of `dir1` is `root`, which leads to `Path(root, dir3)` being recursively evaluated. But evaluation of `Path(root, dir3)` is precisely the computation which initiated this chain of events. Unless this circular evaluation of `Path()` is detected, the evaluation of `Path(root, dir3)` does not terminate.

The initial concern in RCMS was to avoid non-terminating computations such as this. Thus, full caching of predicate evaluation results was not implemented. Instead, a stack structure is used to record that a particular evaluation is *active*, meaning that it is still being computed. If a subsequent attempt is made to evaluate the same predicate with the same arguments, a record of that evaluation can be found on this stack.

In fact, each composite object maintains its own stack for recording active predicate evaluations. The behaviour `checkActiveAndAdd()` lies at the heart of detecting cyclic predicate evaluations. It performs two functions. First, it checks whether the predicate evaluation specified by its arguments (the name of the predicate, and the values of the parameters passed to that predicate), is currently on the stack. If so, it returns *true*. Otherwise, it adds to the stack a record of the predicate evaluation. The behaviour `removeLast()` pops a record off the stack.

Each behaviour which represents a predicate begins with an invocation of `checkActiveAndAdd()`. If `checkActiveAndAdd()` returns *true*, then the predicate evaluation is active, and continued evaluation of the predicate will be non-terminating. Consequently, the predicate returns *false* if the current evaluation is found to be active. Note that the meaning of a return value of *false* is that the predicate being evaluated could not be proven *true*. This is similar to the semantics of query failure in Prolog [51]. If `checkActiveAndAdd()` returns *true*, then the body of the predicate is evaluated. Each behaviour representing a predicate ends with an invocation of `removeLast()` which pops its evaluation record off the stack, indicating that it is no longer active.

The use of a predicate evaluation stack places limitations on RCMS. First, because there is only a single stack, it is not possible for evaluation of predicates to proceed in parallel. Second, recursively defined predicates which rely on side-effects do not work because recursive evaluations which appear to cause non-termination are prevented. RCMS cannot know that such an evaluation may actually terminate as a result of side-effects caused by invocations performed within the body of the predicate. The former limitation could be overcome by providing predicate evaluation stacks on a per-thread basis. The latter limitation should not be too distressing because a recursive predicate designed to rely on side-effects is badly designed.

Code Generation for Logical Expressions

The final component of the compiler is the logical expression compilation routines. As has been noted already, several other portions of the compiler use these routines to generate code which is then integrated into the code which they produce. For example, the body of the `verify()` behaviour consists of almost entirely of code compiled from the set of constraints given in the specification clause.

As with most compilers, each logical expression is represented internally as a parse tree. Each node of the tree represents a particular operator, and the children of that node represent its operands. In the RCMS, each node is also tagged with a variety of information, for example, a reference to a symbol table which describes local variables, representations for the generated code corresponding to the sub-expression rooted at that node, flags providing context information etc. The parse tree is traversed several times, with each traversal adding additional information to the nodes of the tree. The final traversal generates code.

Many aspects of code generation are straightforward. For example, code generation for an invocation consists of taking the sub-expression for the object being invoked on and the sub-expressions representing the arguments, and emitting the Raven code to perform an invocation. Consequently, the description of the code generation process for expressions is limited to those constructs which require significant processing or use interesting implementation techniques.

A reasonable place to begin is to consider code generation for the primitive factors out of which an expression is constructed. Most of these are straightforward: integer constants, string constants, predicate evaluations, variable references, and static set constructors. Dynamic set constructors, on the other hand are quite interesting. A dynamic set constructor looks like:

```
all ident : expression
```

Intuitively, the expression on the right hand side of the colon is a membership predicate for the set being constructed. However, to completely define this construct it is necessary to define the domain over which the membership predicate is evaluated. As noted in Section 4.2, the domain for a set constructor (and also for quantifiers) is the change set of the composite. Unfortunately, the change set for a composite is invariably very large, hence, reasonably efficient evaluation of set constructors demands some form of optimization. Fortunately, there is a wide class of set

constructors for which a relatively straightforward optimization exists. Consider the set constructor:

```
all X : X in foo
```

Clearly this evaluates to a set containing all of the objects in `foo`. The naive implementation of set constructors would generate this set by iterating over the elements of the change set and determining for each object whether it was a member of `foo`.

Unfortunately, the change set for even the simplest composite objects is very large. To estimate the size of the change set for an object, consider the object to be the root of a tree. Each of its instance variables which refers to another object is a branch leading to a child node. The only leaf nodes are those whose instance data consists solely of primitive values such as integers. Of course, the resulting structure is not really a tree because object references may introduce cycles or short cuts. Nonetheless estimating the size of this tree provides a rough idea of the size of the change set. It is well known that the number of nodes in complete tree of this sort is: b^d-1, where b is the branching factor and d is the depth of the tree. Of course, neither the branching factor nor the depth are fixed in this case, but we can see that the number of nodes is exponential with respect to the depth of the tree and thus can grow very quickly. These rough calculations are supported by our experience with the system, in which even simple composites such as the directory tree have a change set containing in excess of 500 objects.

Given that iterating over the change set is likely to be highly inefficient, an alternative needs to be found. For this particular example, an obvious improvement is to simply iterate over the contents of `foo`. The following section reveals how this sort of optimization can be used for a wide variety of set constructors.

Optimization of Iterable Dynamic Set Constructors

The optimization is possible because the particular set constructor used in the example is expressed in terms of other pre-existing sets. In the example, the pre-existing set referred to is `foo`. In more general cases, the set constructor may be a function of one or more sets which are referred to in the membership expression, in which case the membership function is said to be *iterable* (because the set constructor may be evaluated by iterating over pre-existing sets rather than requiring the computation of the change set).

Theorem: A logical expression in disjunctive normal form is iterable with respect to an unbound variable `X` if each of its conjunctive clauses is iterable with respect to `X`. A conjunctive clause is iterable with respect to `X` if at least one of its factors is a non-negated use of the `in` operator with the variable `X` on its left hand side.

Proof: There are two parts to the proof. In the first part, the sets referred to in the expression are used to construct a *candidate set.* In the second part, the candidate set is shown to be equal to the set specified by the dynamic set constructor associated with the iterable expression.

The construction of the candidate set is straightforward. Take each conjunctive clause and separate it into two expressions, one of which is the non-negated use of the `in` operator with X on the left-hand side and the other of which is the remaining portions of the clause. The value on the right hand side of the `in` operator must be a set. Label this set S_i for the i^{th} clause. Label the remainder of the clause $T_i(X)$. Since X is a free variable in the expression it becomes a parameter to T_i. Thus, each conjunctive clause C_i is of the form:

$$C_i = x \in S_i \wedge T_i(x)$$

Assuming that the expression is composed of n conjunctive clauses, then the candidate set is:

$$\bigcup_{i=1}^{n} \{x \| \ C_i(x)\}$$

To demonstrate that the candidate set and the value of the set constructor are identical is really very straightforward. Let Δ be the change set of the composite, then the value of the set constructor is:

$$\{x \| \ x \in \Delta \wedge (C_1(x) \vee C_2(x) \vee \ldots \vee C_n(x))\}$$

Clearly, if an object X is in the set constructor then it is in the candidate set as it must satisfy one of the C_i and that makes it a member of one of the sets whose union is taken to produce the candidate set. The only remaining task is to demonstrate that a member of the candidate set is a member of the set constructor. If an object X is a member of the candidate set then it must be an element of one of the sets whose union is taken to produce the candidate set. Thus, X must satisfy one of the C_i. All that is needed to show that it must be in the set constructor is to demonstrate that it is a member of the change set and since the change set represents all objects which the composite could ever obtain a reference to, it is obvious that X is in the change set. [end of proof]

The proof demonstrates under what circumstances this optimization is useful, and how the compiler goes about generating code for set constructors with iterable membership expressions. As noted previously, the default implementation for a dynamic set constructor is to iterate over the change set and evaluate the membership expression. The time required for this is $N{\cdot}D$, where N is the number of elements in the change set and D is the time taken to evaluate the membership expression for a single object. For set constructor with an iterable membership expression, each S_i is iterated over. For each object X generated by the iteration of S_i, $T_i(X)$ is evaluated. If it eval-

uates to *true* then X is added to the value of the set constructor. If D_i is the time taken to evaluate $T_i(X)$ and N_i is the number of elements in S_i then the time to generate an iterable set constructor is $\sum N_i \cdot D_i$. From observation we know that it is almost always true that $N > \sum N_i$. Since each $T_i(X)$ is a part of the membership expression, it follows that $D > \sum D_i$. Thus,

$$N \cdot D > \sum N_i \cdot \sum D_i > \sum N_i \cdot D_i$$

and, in fact, the size of the change set is frequently very much larger than the sizes of the sets used in an iterable expression, so the time savings can be considerable. For example, the directory tree composite used in the sample interactive session in Section 4.4.1 has a change whose size is approximately 500, while the directory tree itself contains only 4 nodes.

Code Generation for Iterable Expressions

Code generation for iterable expressions differs substantially from normal code generation. A separate traversal of an expression parse tree is made to determine whether an expression is iterable or not. This traversal also marks the nodes with a flag indicating that they are part of an iterable expression. During the code generation traversal this flag alters the form of code generation used. Normally, code generation produces two pieces of code for each node in the tree. The first code segment is Raven code, which when evaluated, yields the value of the sub-expression rooted at the node. The second code segment, called the preamble, is a series of Raven statements which must be executed prior to evaluation of the sub-expression value. When an expression is marked as iterable, four code segments are required. The first pair specify the value and preamble for the set being iterated over. The second pair specify the value and preamble for the filter expression which must be applied to each item of the iteration. It should be noted that this scheme is not fully implemented in the current RCMS compiler. Disjunction is not currently handled, that is, iterable expressions must consist only of a conjunction of factors.

Code Generation for Quantification

Both types of quantification are similar to dynamic set constructors in that they require a domain of evaluation. As with set constructors, the default domain of evaluation is the change set of the composite. If the filter expression of the quantifier is iterable with respect to the quantified variable, then a set containing the domain may be created as outlined for iterable set constructors. In either case, code generation is similar. The set containing the domain is iterated over. Each item of the iteration is assigned to a local variable which represents the quantified expression, and then the body of the expression is evaluated. For universal quantification, evaluation of the body must yield *true* for all items in the iteration. As soon as an item is found for which evaluation of the body yields *false*, the iteration is terminated and then value of the quantified expression is set to *false*. For existential quantification only a single item of the iteration need be located which causes the body to evaluate to *true*. As soon as such an item is found, the iteration is terminated and the value of the quantified expression is set to be *true*.

As should be clear from the implementation, one cannot establish absolute bounds on the time taken to evaluate a quantified expression. This is a result of several factors. First, the size of the domain of quantification determines the maximum number of times the iteration loop is performed. Most composites consist of a reasonably small number of components, and it is usually reasonable to use a filter expression which limits the domain of quantification to the component set. Consequently, the quantified expression can usually be evaluated in a small number of iterations. However, each time through the iteration loop requires evaluation of the body of the quantified expression. This may consists of predicate evaluations, further quantified expressions, invocations on Raven objects, etc. Thus the total time required to evaluate a quantified expression is greatly affected by the times required to complete the Raven invocations used to extract state information from objects in the composite.

Code Generation for the `in` Operator

Although the `in` operator is used to define iterable expressions, it can also be used as a simple predicate. In order to maximize flexibility, the `in` operator works with any Raven object which supports iteration. Code generation for the `in` operator thus consists of iterating over the aggregate object specified by the right hand side of the `in` operator, checking to see if any of the items obtained are identical to the object on the left hand side. This implementation is non-optimal if the object on the right hand side supports a more efficient membership test. For example, class `IdSet` is implemented using a hash table and supports the behaviour `isMember()` which allows constant time checking of membership in the set. A better implementation of the `in` operator would take advantage of this efficiency.

Order of Evaluation

The only other element of code generation meriting discussion is support for McCarthy evaluation. McCarthy evaluation reduces the amount of work done when evaluating logical connectives. Expressions are evaluated left to right. As soon as it is possible to determine the value of an expression, evaluation is terminated and a value returned. For example, if the evaluation of the first term of a disjunction yields *true*, then it is not necessary to evaluate the second term, since it is already known that the value of the disjunction is *true*. RCMS supports McCarthy evaluation. The only significance to this is that the representation of generated code as pairs of code segments representing a preamble and a value requires that some care be taken. For example, when generating code to evaluate a disjunction, the compiler must emit the preamble for the left hand side, a test of the value of the left hand side, and then a segment of code which is only executed if the left hand side yielded false. This code consists of the preamble and value for the right hand side. The key point is that preambles cannot be executed until such time as it is known that their associated values are required.

5.2 The RCMS Class Library

There are, in fact, only two Raven classes which are specific to the RCMS. The first is class `RCMS` and the second is class `Composite`. The former encapsulates various RCMS management functions, while the latter represents the common interface shared by all composite objects.

5.2.1 The RCMS Class

For each Raven virtual machine, there is a single instance of class `RCMS` which is created during system initialization. Figure 23 contains the declaration of this class. It does not currently sup-

```
class RCMS controlled
    {
    behav notifyRemoteFailure(src, dest:Object, methodName:String) no lock;
    behav pollComposite(aComposite:Composite, period:Int) write lock;
    behav monitorFailure(aComposite:Composite) write lock;
    constructor();
    }
```

Figure 23 – Declaration of class RCMS

port a large number of operations. Behaviour `notifyRemoteFailure()` is part of the implementation mechanism used to monitor composite objects for failure of inter-object links. It is discussed more fully in the next section. Behaviours `pollComposite()` and `monitorFailure()` are used to specify the type of monitoring that should be used by a composite.

An invocation of `pollComposite()` signifies that `aComposite` should be periodically polled and validated. The time between polls is `period`, which is specified in the same units as the system clock. Records indicating which composites are to be polled are maintained on a delta list sorted according to the next polling time. When it is allocated, the RCMS object creates a separate thread which periodically checks the delta list for composites which currently need to be polled. The poll is carried out by invoking `checkAndRepair()` on the composite.

Note that the declaration of `pollComposite()` has the keywords `write lock` appended to it. This effectively provides mutual exclusion for the behaviour. This mutual exclusion is necessary because `pollComposite()` modifies the delta list, and simultaneous access by more than one thread would result in inconsistency.

An invocation of `monitorFailure()` signifies that `aComposite` should be validated only when the remote communication facility of Raven detects failure of an inter-object link. In the current implementation this does not require changing the state of the RCMS object. All that is required is that the composite object itself be tagged with an indication that remote failure monitoring is being performed.

5.2.2 The Composite Class

Figure 24 contains the declaration for class `Composite`. All class declarations for placeholder objects created by the RCMS compiler inherit from this class. It specifies the interface which is common to all placeholder objects. Additionally, the full definition of `Composite` provides implementations for several behaviours which are shared by all placeholder objects. As noted in Section 5.1, instance variable `components` contains references for all objects which are currently part of the composite. Instance variable `active` represents the stack of predicate evaluation records which is used to detect recursion loops. Instance variable `currentConstraint` contains the label of the last constraint evaluated by the `verify()` behaviour. Finally, instance variable `monitorType` indicates what variety of monitoring is being used for the composite object. The integer value is used as a set by associating each type of monitoring with a particular bit. Behaviours `verify()`, `updateComponents()`, and `repairComponents()` are part of the common interface of placeholder objects, but their implementations are always provided by the RCMS compiler.

```
class Composite <- Object controlled
    {
    components : IdSet[Object];
    active : List[EvalRecord];
    currentConstraint : Int;
    monitorType : Int;
    behav verify() : Int write lock;
    behav updateComponents() write lock;
    behav repairComponents() write lock;
    behav checkAndRepair() write lock;
    behav checkActiveAndAdd(...) : Int private;
    behav genTransitiveClosure(root:Object) : IdSet[Object] private;
    }
```

Figure 24 – Definition for class Composite

Behaviour `checkAndRepair()` is the usual means for carrying out the validation/repair cycle for a composite object. Its implementation consists of a loop, the body of which:

(1) Invokes `updateComponents()` to ensure that the components set for the composite is correct.

(2) Invokes the `verify()` behaviour to determine whether the composite satisfies the constraints of its specification clause. If so, the loop is terminated and `checkAndRepair()` returns to its caller.

(3) Invokes the `repairComponents()` behaviour.

Currently, no attempt is made to detect non-termination of the validate/repair cycle. A count of the number of iterations through the loop is maintained, but is not used. It is possible that each composite object should have a publicly accessible instance variable indicating the maximum number of iterations allowed. This variable could be set by the programmer, based on knowledge of how the repair scripts operate. The presence of the keywords `write lock` on the declaration of `checkAndRepair()` ensures that only one `Thread` can engage in the validate/repair cycle for a particular composite object. As was noted in Section 5.1, preservation of the consistency of the predicate activation stack requires this. Furthermore, consistency between the

components set computed by `updateComponents()` and the view of the composite seen during the execution of `verify()` requires that the state of the composite remain stable during `checkAndRepair()`.

The only behaviour which remains is `genTransitiveClosure()`. This behaviour computes the change set for a given object. Its implementation consists of a call to a run-time routine written in C responsible for carrying out the computation. The motivation for using a C language routine reveals some interesting features of Raven. The basic form of the routine for computing a change set is quite simple:

(1) Add the current object to the change set.

(2) For each object reference in the instance data of the current object check to see whether it is in the change set. If not, compute its change set. Add the contents of the latter to the change set being computed.

(3) Return the change set to the caller.

Since the Raven compiler knows the types of the instance variables for each class that it compiles, it is possible to have it automatically generate a behaviour to compute the change set for a particular type of object. However, this is somewhat inefficient. First, each class has additional code generated for it. Second, actual computation of the change set would involve many levels of recursive invocations. Raven addresses this problem by providing C level routines which provide information about the type of data stored in any instance variable of any class. Thus, it is possible to write a single C routine which takes an object, looks up the types of its instance variables, and then, for each instance variable which is a reference to another object, recursively calls itself to compute the change set. This is the implementation used by the C routine which `genTransitiveClosure()` calls. The unfortunate aspect of this type of implementation is that it exposes a number of internal details of the Raven system to the C routine which computes the change set. In particular, it must handle remote objects differently from local ones, since

their instance data cannot be accessed in the straightforward manner used for local objects. This makes maintenance of the Raven system difficult, as changes in the representations of objects must be propagated through to any C level routines which access objects directly rather than through the invocation mechanism.

5.3 Implementation of Roll-back Repair

Provision of recoverable repair for composite objects entailed changes in Raven's support for recoverable objects. Description of these changes requires at least a basic understanding of how recoverability is provided in Raven.

5.3.1 Object Properties in Raven

The basis for recoverability, and, in fact, other properties such as persistence and locking, is Raven's flexible invocation mechanism. The invocation mechanism allows wrappers to be placed around the actual execution of behaviour code. Each wrapper is composed of two segments of code, one which executes before the actual code of the behaviour, called a *pre-handler*, and another which executes afterward, called a *post-handler*. When an invocation is performed, the invocation routines check to see what properties the object being invoked on has. The pre-handlers for these properties are executed, then the code of the behaviour, and then the post-handlers.

5.3.2 Semantics of Recoverability

The semantics of recoverability in Raven are relatively straightforward. Any behaviour of a recoverable object may execute the `restore` statement. Assuming that the behaviour had made no invocations on objects other than itself, the effect of this would be to restore the instance variables of the object to the values they had on entry to the behaviour. If execution of the behaviour caused invocations on other recoverable objects to occur, then the `restore`

statement would have the effect of rolling-back any changes made to these objects as well. Ignoring for the moment the effects of concurrency and assuming that all objects are recoverable, the `restore` statement allows the programmer to roll-back the state of the world to the time at which the behaviour was entered.

In Raven, an attempt has been made to separate the issues of recoverability, persistence, and concurrency control. This provides both flexibility and the possibility of increased efficiency. It also provides the programmer with the proverbial "length of rope" with which he may hang himself. For example, if the programmer can guarantee that a collection of objects will never be accessed by more than one thread concurrently, then he may choose to make those objects recoverable, but not concurrency controlled. The advantage of this is that the overhead required to provide concurrency control is not incurred for invocations on those objects. However, if the programmer has made an error, and concurrent access does occur, the likely result is disaster. While one thread executes a `restore` statement, another thread may be altering the contents of various objects. Objects whose states are inconsistent with one another result from this sort of concurrent access.

5.3.3 Implementation of Recoverability

The lack of interaction between support for persistence and support for other object properties simplifies the recovery mechanism. The basis for recoverability in Raven is the maintenance of *before images* of modified objects. A before image is a copy of an object's instance data made prior to the execution of a behaviour on that object. These before images are maintained in a data structure which represents the call tree for a particular thread. Each node in the call tree represents an invocation of a behaviour on an object. The children of a node correspond to invocations on other objects made during the execution of the behaviour represented by that node. Threads of control in Raven are represented by Raven objects of class `Thread`, and a reference

to the call tree data structure for a particular thread is stored in the instance data of its corresponding `Thread` object.

When an invocation on a recoverable object is made, the recovery pre-handler is executed. It is supplied with arguments indicating the target of the invocation, the behaviour being invoked, and information concerning that behaviour, such as the number of arguments which it takes. The pre-handler allocates a new node for the call tree, adds it to the tree, and stores in the `Thread` object a reference to this new node as the current location in the call tree. If the invocation results in any changes to the instance variables (this information is produced by the compiler and made available to the pre-handler), a copy of the instance data of the object is made and stored in the call tree node. Control then returns to the invocation manager which executes the code of the behaviour, after possibly executing pre-handlers for other object properties.

Following execution of the behaviour code, and possibly the execution of other post-handlers, the recovery post-handler is executed. It changes the reference to the current call tree node stored in the `Thread` object to refer to its parent node. Moving up one level in the call tree corresponds to returning from a behaviour. If the current call tree node does not have a parent, then the behaviour which it corresponds to is a top-level recovery block. Since that behaviour has now completed execution, there is no further need for the recovery information stored in the call tree, and it is thrown away.

Execution of the restore statement within the body of a behaviour is translated by the Raven compiler into a call to the `recoveryRestore()` routine of the recovery manager. This routine obtains the pointer to the current location in the call tree from the `Thread` object. This node contains the before image of the object for the currently executing behaviour. Its children represent invocations already performed by the behaviour. In order to restore the state of all objects modified by the current behaviour, a depth first, right-to-left, post-order traversal of the

sub-tree rooted at the current node is performed. For each node in the traversal, if a before image exists, it is substituted for the current values of the corresponding object. The nature of the traversal ensures that the most recent invocations are restored first and the oldest last. Thus, if a given behaviour makes more than one invocation on a given object, the overall result is that the before image for the oldest invocation is the one which remains in use following the recovery.

There is one final point to be made with respect to the current support for recoverability in Raven: it does not yet work in a distributed environment. Part of the reason for this lies in the manner in which `Thread` objects are implemented. As noted above, information related to recovery is stored within the Raven `Thread` object. When an invocation crosses machine boundaries, the identity of the `Thread` object changes. Although in an abstract sense there is only one thread of control, which has moved from one machine to another, there are two or more Raven `Thread` objects which represent it. The current recovery manager design assumes that the entire call tree for a thread is accessible via the `Thread` object. Clearly, when an invocation crosses machine boundaries, this assumption is violated. One solution to this problem is an implementation of the recovery manager which correctly manages the multiple instances of Thread. However, this implementation also requires changes to the Thread class and to the remote invocation manager of Raven. A more elegant solution is to modify the handling of `Thread` objects in Raven such that `Thread` identity is preserved across machine boundaries. Doing so provides a level of location transparency for implementers of property handlers which interact with Raven threads (for example, the concurrency control manager).

5.3.4 Changes to Raven to Support Roll-back Repair

The provision of roll-back repair for composite objects requires changes to the recovery post-handler. When the post-handler detects the exit from a top-level recovery block, a call is made to an RCMS support routine called `configCheck()`. This routine examines each node in the

call tree. For each node, it checks to see whether the object represented by that node is currently a component of any composite object. Recall that the `updateComponents()` behaviour of any composite object installs back-pointers from components to the placeholder object so that this test is easy to perform. For each composite to which it belongs, the `updateComponents()` and `verify()` behaviours are invoked. If any invocation of a verify behaviour fails, `configCheck()` returns false to the recovery post-handler, which, in turn, rolls-back the entire recovery block.

Note that the use of roll-back recovery in RCMS offers the same flexibility and pitfalls as recoverability for Raven objects. Namely, it is possible to create a recoverable composite out of objects which are not concurrency controlled. Concurrent access to the components of such an object can lead to unpredictable results. For example, a recovery block may be rolled back due to changes made by another thread.

The current implementation of roll-back recovery is complicated by side-effects caused by the execution of the `updateComponents()` and `verify()` behaviours. The process of traversing the call tree, following back-pointers to placeholder objects, and invoking the `updateComponents()` and `verify()` behaviours occurs within the context of the recovery block which is being checked. From the point of view of the implementation, these modifications which occur during verification appear as new nodes in the call tree. In fact, checking one node in the tree introduces several other new nodes. As a result, the straightforward implementation for checking all nodes in the tree is non-terminating. That is, a loop which simply iterates over the nodes of the tree performing the standard checking operation for each, never terminates because nodes are added to the call tree more quickly than they are processed.

This problem is addressed by recognizing that invocations performed by RCMS within the context of the recovery block are not significant with respect to validation. These nodes of the call

tree are skipped. For example, access to the back-pointers from elementary objects to placeholder objects is done using standard Raven invocations and, hence, produces new nodes in the call tree. However, since these nodes correspond to RCMS activities, they are skipped when RCMS processes the call tree.

There is one last aspect of recoverable composites which needs to be addressed. The objects in the call tree for a recovery block are those directly accessed during the recovery block. It is possible that a change to such an object could invalidate a composite without that object actually being registered as a part of that composite. Not being a part of the composite, such an object lacks the back-pointer to the placeholder object for the composite which it affects. Consequently, when RCMS processes the call tree for the recovery block, it is unaware of the need to validate that composite.

For example, suppose that one of the instance variables of a composite is a set of objects, each of which is assigned a cost. Suppose, in addition, that the set supports a behaviour to compute the average cost of the items which it contains and that the composite has a constraint stating that this average cost may not exceed a particular value. Because the instance variable for the set lacks the indirect attribute, the contents of the set are *not* considered to be components of the composite. An alteration to the cost of one of the items in the set results in a node being added to the call tree. But when RCMS processes the call tree it ignores that node because the item is not a component. Consequently, the composite's constraint may be violated. This problem is called *undetected modification.*

In this example, it is fairly obvious that the programmer responsible for designing the composite has made a mistake. That is, the instance variable for the set should have been given the indirect attribute so that the items contained in the set became components of the composite. However, responsibility cannot always be placed on the shoulders of the programmer in this way. Raven is

object oriented, and one of the consequences of this is that the internal structure of objects is not visible to their users (this is referred to as *encapsulation*). Thus, it may be difficult for the programmer to explicitly identify all the objects which should be components of a composite. One solution to this problem makes use of a new feature of Raven, and this is discussed in Section 6.2. A second solution is to make all objects in the change set of the composite components. However, this has the disadvantage of being inefficient. An entirely satisfactory answer to this problem is an area for further investigation.

5.4 Implementation of Object Monitoring

The recovery support system in Raven provides a convenient point to attach the RCMS routines which handle roll-back repair. Detecting changes in objects and repairing those changes are both effectively handled by the recovery support system. The call tree provides the set of objects modified by a thread, the exit from a top-level recovery block provides an appropriate time at which the RCMS can intercede, and the recovery mechanism itself provides a straightforward way to repair any invalid composites. When roll-forward repair is used, these issues are not so conveniently dealt with. RCMS needs to know what sort of events should cause validation/repair of a composite. That is, RCMS requires an explicit monitoring mechanism.

5.4.1 Implementation of Polling

The first sort of monitoring used in RCMS is polling which was discussed in Section 5.2. Another simple monitoring mechanism causes validation/repair whenever the state of a component changes. This form of monitoring relies on Raven's property handlers. RCMS introduces a new property into Raven called `monitored`. When a monitored object changes state, the composites to which it belongs carry out validation/repair.

5.4.2 Implementation of Real-Time Monitoring

When a composite object is tagged as using real-time monitoring, the `updateComponents()` behaviour not only installs back-pointers into component objects, it also gives them the `monitored` property. The pre-handler for the `monitored` property makes a copy of the instance variables of the object. The post-handler for the `monitored` property compares the current values of the instance variables against the stored values. If any of the values differ, then the back-pointer to the placeholder object is followed, and the `checkAndRepair()` behaviour is invoked.

This form of monitoring suffers from two problems. First, it is subject to the same undetected modification problem as roll-back repair, namely, changes to objects which are not components may lead to invalidation of the composite. However, this difficulty is amenable to the same sorts of solutions as those used for roll-back repair. Second, in many circumstances this form of monitoring is overly sensitive. A composite object subject to this type of monitoring cannot enter an invalid state, even temporarily. For example, suppose that a programmer is using a composite object which supports the functions of `make`. If the granularity of the editing behaviours on the source code objects is at the level of single blocks of code, then RCMS would attempt re-compilation after every change to a block. Suppose that the programmer changes the name of a function. His intention is to then find all references to that function and change the calls as well. However, upon completion of the first edit, an overly eager RCMS would begin re-compilation, which would, of course, fail due to the inconsistencies.

The simplest solution to this problem is to place the burden of monitoring on the user of the composite object. So, just as the programmer must run `make` when he wishes to re-build an application, the user of a composite object could signal RCMS when validation/repair of a composite is desired. In fact, this is easily accomplished in the current implementation, the user sim-

ply invokes the `checkAndRepair()` method on the placeholder object. Certainly, this ensures that RCMS will not interfere with the programmer, but it makes RCMS passive, and, thus, eliminates the guarantee that composite objects always satisfy their specifications.[1]

5.4.3 Monitoring via Remote Invocation Failure

Another type of event which makes a useful trigger for composite validation/repair is exceptions and failures. The current implementation of RCMS supports triggering on the occurrence of a remote invocation failure. This mode of monitoring is used in the distributed filter example of Section 4.4.4. Prior to delving into Raven's remote invocation manager, a brief digression into the implementation of links is required.

The condition intended to trigger validation/repair of the distributed filter composite is the failure of the inter-object link connecting the output of one filter to the input of the next. However, inter-object links are implemented on top of the normal invocation mechanism of Raven. Declaration of an outbound link in Raven causes the compiler to create a hidden instance variable representing the state of the link. Each outbound link is associated with a list of records describing the link's connections. Each record contains the target object of a connection and the name of the inbound link being connected to. For example, the following code:

```
class SampleOutLink
    {
    link out dataOut(Int);
    }
```

causes the compiler to create an instance variable named `_targets_dataOut`. This variable is of type `List[LConnection]`. Each `LConnection` in the list contains the capability of

1. The sensitivity offers ample opportunity for extending RCMS with improved monitoring techniques. For example, monitoring could be tied to particular sequences of invocations, or invocations with parameters which satisfied certain requirements. None of these possibilities has been explored, and without further investigation it is difficult to assess which features would be most useful.

an object to which `dataOut` is connected and the name of the behaviour of that object which should be invoked when data is sent along the link.

Similarly, declaration of an inbound link creates a hidden instance variable which records pairs of objects and outbound link names which represent the current connections of that inbound link. In addition, the code associated with an inbound link is used as the body of a hidden behaviour. For example,

```
class SampleInLink
    {
    link in dataIn(p1:Int);
    }
```

causes the creation of a hidden instance variable `_sources_dataIn` and a hidden behaviour `_link_dataIn(p1:Int)`. The hidden instance variable `_sources_dataIn` has type `List[LConnection]`. Each `LConnection` in the list contains the capability of an object from which a connection to the link originates and the name of the outbound link. The body of the link behaviour is supplied by the programmer. The `send()` statement, which is used to transmit data on an outbound link, translates into an invocation on the `System` object which takes each `LConnection` from `_targets_<link-name>` and performs the invocation specified by the object/behaviour name stored in that `LConnection`.

Thus, the failure of an inter-object link which spans a machine boundary is detected within the system as the failure of a remote invocation. This failure is initially detected within the routines which implement the remote invocation protocol (which is similar to many other existing RPC protocols). However, the current implementation of RCMS intercepts the failure at a slightly higher level, in the Raven run-time routine `executeRR()`.

This routine is called by the invocation manager when it detects that the object being invoked on resides on a different machine. It takes the capability for the remote object, the name of the

behaviour being invoked, and the values of the parameters for the invocation and effectively constructs a remote invocation request which is then given to the routine which implements the remote invocation protocol. If the remote invocation cannot be completed, an error indication is returned to `executeRR()`.

Changes to Raven to Support Error Trapping

Support for RCMS is inserted at this point. If an error indication is returned, the routine `HandleFailure()` is called. Basically, its task is to invoke the `notifyRemoteFailure()` behaviour on the RCMS system object. However, a straightforward invocation is not acceptable. When validating/repairing the composite, RCMS wants to see it in a consistent state. In order for this to be true, there should be no other threads of control executing invocations on any components. However, at the time the remote invocation failure occurs, we know that the thread performing the `send()` is executing within a component. Furthermore, that thread is blocked inside `executeRR()` until the failure handling is complete. In order to allow `executeRR()` to complete (and consequently to allow `send()` to return and the thread to complete its invocation on the component), a new thread is created to carry out the invocation of `notifyRemoteFailure()`.

The `notifyRemoteFailure()` behaviour of the RCMS object begins by checking the name of the behaviour being invoked. Failure of a send on a link is indicated by the behaviour being `linkSend()`. If any other behaviour name is found, `notifyRemoteFailure()` simply terminates. Next, for each composite object of which the sender is a component, the `checkAndRepair()` behaviour is invoked on the placeholder object for the composite. Once this is complete, `notifyRemoteFailure()` terminates, and the thread created in `HandleFailure()` is destroyed.

The link manipulation routines of the `System` object also had to be modified to operate correctly with respect to communication failures. Recall that the existence of a connection between an outbound and an inbound link is recorded in both the source and destination objects. Some of this information is redundant in the sense that the set of all existing connections could be described by just the pairs of objects/behaviour names associated with each outbound link. However, some features of links in Raven are difficult to implement without the redundant information. For example the, `numInputs()` behaviour of the `System` object returns the number of outbound links connected to a given inbound link. Without the redundant information associated with the inbound link, this behaviour would be forced to examine every object in the system in order to check for the existence of an outbound link connected to the inbound link in question. Because the implementation of links relies on redundant data, behaviours which modify the state of a link are responsible for maintaining its consistency. Consider, for example, the `unLink()` behaviour. Eliminating a connection involves modifying both the source and the destination objects. At the source object, the destination object/behaviour name pair have to be removed from the list of targets. At the destination object, the source object/outbound link name pair have to be removed from the list of sources. The original implementation of the `unLink()` behaviour would fail if either the source or destination objects could not be accessed. Such an implementation is not suitable for use with the distributed filter composite. Consider the actions taken by the repair script in the event of link failure: the first filter is unlinked from the second followed by establishment of a new link between the first filter and the buffer. However, in this example, the second filter is no longer accessible from the machine on which the first filter and the placeholder object reside. Consequently, the invocation of unLink fails, and repair of the composite is not possible.

The problem was solved by weakening the consistency requirements for link connection data. The ability to transmit data over a link is a function of the connection data maintained at the

source object. In the modified implementation, therefore, the existence of a link is defined only by this data. The data maintained at the destination object is allowed to become inconsistent in the event of failures (communication failures, objects destroyed due to software errors, etc.). Currently the only function whose semantics are affected by this change is `numInputs()`. In the event of failures the value returned by this routine may not be accurate, and, indeed, cannot be made accurate because communications failures may make it impossible to contact the source object of a possible inbound connection in order to verify the state of the connection.

It may well be that these semantics for inter-object links are not appropriate. At the moment inter-object links are really no more than RPC in a convenient disguise. They are not, as their name might suggest, virtual circuits, which have a much stronger definition of connectedness. It is possible that links should offer the stronger semantics of virtual circuits. This was not explored in my thesis work, and represents yet another area in which the design of the configuration management system interacts with the design of the underlying system.

Returning to `unLink()`, its modified implementation only fails if the source object cannot be accessed. Connection data at the destination object is updated if the destination object is accessible. The repair script for the distributed filter works correctly with this new implementation. Repair occurs on the machine on which the placeholder object resides. This is also where the first filter component resides. Despite the fact that the second filter component is no longer accessible, unlinking the connection between the first and second filter succeeds. The first filter is connected to the buffer and the composite returns to an acceptable state.

Of course, the connection data at the second filter has become inconsistent. From the point of view of RCMS, this is not a problem. The second filter is not being used by the composite object, and it should not be visible outside of the composite. When communication between the first and second filter becomes possible again, RCMS validates and repairs the composite. Dur-

ing execution of `updateComponents()`, `updateLinkStatus()` is invoked to re-establish connection data consistency for objects with links.

The manner in which RCMS is integrated into Raven's link support routines creates one further complication. Link failure is not detected by RCMS until such time as a send operation is attempted. At the point where RCMS becomes aware of the link failure, the link support routines have effectively abandoned trying to deliver the data associated with that send operation. Ensuring that this data is not lost entirely could be handled in two ways: (1) Failure of a send operation would cause an exception to be signalled to the application and it would be the application's responsibility to re-transmit the data, (2) The inter-object link implementation would detect the failure of a send and attempt to re-send the data itself. The problem with the first approach is that Raven does not support an exception-handling mechanism. In order to facilitate the second approach, the information regarding the failed send operation is passed along to `notifyRemoteFailure()`. Although this data is not currently used, a re-send facility could be added to `notifyRemoteFailure()`.

Weaknesses of the Exception Handling Mechanism

The current implementation of link failure detection suffers from several weaknesses. The most readily noticed is that detection of a link failure and subsequent validation and repair are extremely slow. It takes several minutes for the distributed filter composite to re-configure itself, for example. The root of the problem is, once again, that RCMS does not become aware of link failure until such time as a send operation fails. The underlying remote invocation protocol does not signal failure until a number of re-tries have been attempted, each of which requires a certain time-out period. In the current Raven system, the time in between starting a send operation and receiving a failure notification is about 15 seconds. This delay is compounded by the manner in which RCMS responds. In attempting to validate the composite, RCMS invariably makes

attempts to invoke behaviours on objects which are no longer accessible. Each of these invocations incurs a further delay. Clearly, these sorts of delays are unacceptable.

The compounded delays can be eliminated by modifying the support for remote invocation. Most remote invocation failures stem from communication problems or failures of entire machines, rather than the failure of individual objects. If data were available on which machines could be reached, the first stage of processing a remote invocation would be checking connectivity to the target machine. Providing such data would not be difficult. A cache of connectivity information would be maintained. For each remote invocation performed, the target machine would be added to the cache. Entries in the cache would be aged and eventually discarded. On a regular basis, the remote invocation system would attempt to contact each machine in the cache. The reachability of each machine would be recorded. To improve the hit ratio for the cache, link creation and deletion events could be propagated down to the remote invocation system. The existence of a link to another machine is a good indicator that remote invocations to that machine will occur.

Extensions to Raven's Error Exception Handling

Another deficiency in the current implementation of RCMS is that the only exceptional event it traps is failure of a link. In order to detect when a failed link could be re-established, polling must currently be used. In the distributed filter composite, for example, one of the constraints uses the `Connectable()` primitive to monitor whether the second filter is accessible. Without periodic validation/repair, the composite would not re-configure when the second filter became accessible again.

Part of the problem lies with the Raven system itself. Exception handling is an area of Raven which has received only scant attention. Currently, most failures in Raven are catastrophic in

that they cause the entire Raven virtual machine to halt. Additionally, the only mechanism for notifying the programmer of errors is via return codes. However, as even a simple example such as the distributed filter composite makes clear, exception handling is very important for the implementation of a configuration management system. The more information which the configuration manager has regarding exceptional conditions, the more specific its responses can be. We close off this chapter by looking at one way in which more information about exceptions could be provided to RCMS.

The approach relies on modifications to the link support system. Each link would be associated with an up/down status. The `System` object would use a separate process to periodically test outbound links originating on its machine and set the status. Changes in link status would be forwarded to appropriate composite objects. Furthermore, link status would be accessible to the programmer via a new method on the `System` object. The programming style used in this type of environment would be somewhat different than that exemplified by the distributed filter composite. In general, failed links would be left in place, rather than removed.

For example, in the distributed filter composite, the output of the first filter would be connected to a simple switch object. This object would have one input and two outputs and a behaviour to switch the flow of data from one output to the other. One output of the switch would be connected to the input of the second filter and the other to the buffer. Under normal circumstances, data would flow from the first filter through the switch to the second filter. However, if the link between the switch and the second filter failed, RCMS would be notified. During validation the state of the link would be examined. The repair script would alter the switch so that data was forwarded to the buffer. Because the link to the second filter would still exist, the link support system would be periodically probing it. If communication to the second filter were restored,

RCMS would again be notified. Validation would check link status, and the corresponding repair script would switch data flow back to the second filter.

Link failure may also result from the destruction of the objects at either end of the link. Communications failure and object destruction require quite different responses from RCMS. When an object is destroyed it is pointless to continue to monitor the connection to that object. Communication to that object will not be restored. The proper response is to replace the object and establish a new connection to it. In order to make this response, RCMS requires notification of object failure.

As it happens, detection of object failure in the current implementation of Raven is very difficult. Raven has no explicit means for destroying objects. Memory is reclaimed through garbage collection which operates at the level of C memory allocations and thus is unaware of when it is re-claiming storage associated with Raven objects. Thus, it is difficult to outline a proposal for providing this service for RCMS. The purpose in raising the issue is to draw attention to the impact that system design has on configuration management. The lack of exception handling facilities in Raven limits the configuration management services which RCMS can provide. In Chapter 6, these types of issues are explored more fully and several significant extensions to RCMS are presented.

5.5 Summary

This chapter has demonstrated how the various pieces of the Raven Configuration Management System are implemented. Composite objects are translated into Raven objects by the RCMS compiler. Each composite object is, thus, represented in the system by a corresponding placeholder object. This placeholder object contains behaviours which compute the set of components of the composite, evaluate the constraints of its specification clause, and carry out the

repairs specified in its repair clause. Changes to the underlying Raven system provide the monitoring services which are required to make RCMS an active configuration management system. Three forms of monitoring are implemented in RCMS: polling, real-time, and error trapping. Although these three mechanisms serve the same fundamental purpose, that is, they allow RCMS to detect changes to composite objects on an ongoing basis, their implementations are quite different. Polling uses a simple timer mechanism, real-time monitoring required modifications to Raven's invocation mechanism, and error trapping required adding exception handling services to Raven.

Through focussing on the details of the implementation we identify problems which hamper the performance and utility of the system. The section addressing code generation for dynamic set constructors, shows that a naive implementation based on iterating over the change set of the composite is very inefficient. Consequently an optimization for *iterable* set constructors is presented. The section on real-time monitoring reveals that overhead for this sort of monitoring is quite high, which prompts further exploration of monitoring techniques in Chapter 6. Finally, the limitations of current error trapping scheme point out that improving fault tolerance through the configuration management system demands better exception handling facilities than are currently provided in Raven.

CHAPTER 6

Extensions to RCMS

Chapters 4 and 5 presented a description of the current status of RCMS. Development and use of RCMS has highlighted a number of challenges, both for configuration management in general, and RCMS in particular. This chapter examines a number of these challenges, discusses new perspectives gained via RCMS, and outlines possible solutions.

6.1 Component Set Computations

Configuration management begins with identification of the objects that will be managed. The simplest identification scheme relies on explicit specification of the objects which are components of a configuration. The set of targets and dependencies in a makefile is an example of explicit identification. RCMS provides an approach to identification which is both novel and powerful.

In RCMS, a configuration is represented by a composite object. The objects that constitute the configuration are specified via the instance variables of that composite. By using indirect instance variables and dynamic set constructors it is possible for the components of a composite object to vary over time. For example, the directory tree composite of Section 4.4.1 is defined such that its components set always contains all of the directory nodes of the tree.

The current implementation of RCMS always re-computes the components set prior to validation. This is not strictly necessary. If the instance variables of the composite are given initial values and not modified thereafter, then the components set computation need only occur when the composite is created. Assuming that code for initial computation of the components set was emitted as part of the `constructor()` for the composite, the `updateComponents()` behaviour could have an empty body. Complete elimination of `updateComponents()` is problematic since other parts of RCMS assumes that all composite objects support it.

Components set computations can be divided into three groups based on the amount of work required to update them:

(1) static: the components set does not change over time (as in the example of the previous paragraph). Components set computations need only be carried out once when the composite is created.

(2) iterable: the components set changes over time, but can always be derived simply by iterating over objects accessible from within the composite. The components set needs to be re-computed prior to validation.

(3) general: the components set changes over time. To compute the components set requires computation of the change set of the composite. As with an iterable components set, re-computation is required prior to validation.

6.1.1 Static Components Sets

For a components set to be static, the instance variables of the composite must not be public (ref. Section A-3.0), not appear on the left hand side of a temporal assignment, and not be modified by repair scripts. If an instance variable has the `indirect` attribute, then it cannot be initialized except by a set constructor. These restrictions ensure that the value of a composite's instance variables will not change once the composite has been created and initialized.

Disallowing public instance variables guarantees that no external access will change the value of the instance variable. Note that this condition is overly restrictive. RCMS merely needs to ensure that the instance variables are read-only, but Raven does not currently support this feature. The prohibition of temporal assignments is clear: temporal assignments are specifically intended to allow for instance variables to be updated. Similarly, if the instance variable occurs on the left hand side of an assignment statement in a repair script, it is unlikely that its value is stable.

The final restriction, on instance variables with the `indirect` attribute, is more subtle. The components set is computed by iterating over the *contents* of an `indirect` instance variable. If the contents of the instance variable changes, the components set changes as well. If an `indirect` instance variable were initialized from a `constructor()` parameter, or the result of an invocation on another object (via the initial assignment operator), the possibility remains that a reference to it is held outside of the composite. Its contents could be modified following initialization of the composite. The only values for which it is possible to guarantee there are no external references are those returned by set constructors.

6.1.2 Iterable Components Sets

There is only one requirement for a components set to be iterable. For each temporal assignment whose right hand side is a dynamic set constructor, the membership expression for that set constructor must itself be iterable (ref. Section 5.1). The term iterable is perhaps slightly misleading because an iterable components set may not actually involve any iteration. For example, a composite with no indirect instance variables, but with some public instance variables is classed as iterable. A key feature of an iterable components set is that it is not static, that is, it has to be re-computed prior to validation. The term iterable is used because the most expensive operation used in re-computing the components set is iteration.

The restriction on temporal assignments arises because the computations associated with a temporal assignment occur in `updateComponents()`. If a temporal assignment uses a dynamic set constructor which is not iterable, then the change set for the composite has to be computed and filtered through the membership expression. Thus, a composite with an iterable components set is preferable to one with a general components set since the computation of a general components set is considerably more expensive.

6.1.3 General Components Sets

A general components set is one which must be re-computed prior to validation and which requires computation of the change set. Computing the change set is an expensive operation. In general, the size of the change set grows exponentially with respect to the number of instance variables of the composite. The directory tree composite, for example, uses a general components set to compute the set of nodes of the directory tree.

Experience with RCMS has shown that the time taken to compute the change set of a composite is generally unacceptable. Fortunately, there are techniques through which it is possible to avoid

computing the change set in some circumstances. Consider the directory tree composite. The change set is required to properly evaluate the temporal assignment:

```
tree := all X : X.instanceOf() == DirNode & Path(root, X);          [a]
```

where the `Path()` predicate is:

```
define Path(X, Y) <- Y in X |                                       [b]
  there exists Z [Z in X] (Z.instanceOf() == DirNode) & Path(Z, Y);
```

The membership expression of the set constructor given in [a] contains a restriction, that `X` should be an instance of `DirNode`, which is not strictly necessary. It is simply a feature designed to speed up computation of the set constructor from the change set. The definition of `Path()` given in [b] already guarantees that if `Path(root, X)` is true, then `X` is an instance of `DirNode`. However, most elements of the change set for the directory tree composite are not `DirNodes`. By first checking that `X` is a `DirNode`, many unnecessary evaluations of `Path()` are avoided.

Recall that the change set is computed by recursively following all object references originating from instance variables of the composite and gathering the objects visited into a set. Essentially, this is a depth first search with provisions to avoid non-termination due to cyclic references. However, the `Path()` predicate itself effectively defines a much more refined depth first search which is far less expensive to carry out. Intuitively the `Path()` predicate could be compiled into a recursive routine to compute the set specified by:

```
all X : Path(root,X);
```

Figure 25 contains the definition of a Raven behaviour which does this. The behaviour takes two parameters. The first, `res`, is a set which is used to accumulate the value of the set constructor. The second, `start`, is the `DirNode` at which the search should start. Thus, in order to compute the desired set constructor, an empty set is allocated and the invocation `gen-`

```
behav genPath(res, start)
{
res.add(start);
i = root.getIterator();
while (!i.done())
    {
    var curr : Object = i.next();
    if (curr.instanceOf() == DirNode && !res.isMember(curr))
       genPath(res, curr);
    }
}
```

Figure 25 – Behaviour to Directly Compute a Set Constructor

`Path(answer, root)` is made (assuming that the empty set has been assigned to a variable `answer`). The behaviour `genPath()` is useful because it computes the components set of the directory tree composite without requiring the change set to be computed. Because `genPath()` is a much more restricted form of search than computing the change set, the components set of the directory tree is computed much more efficiently.

Such a transformation may be intuitive, but the important question is whether it is possible to mechanize the process such that the RCMS compiler can recognize set constructors amenable to the transformation and generate the Raven behaviour for computing the sets. The answer to this question is yes, at least for predicates which meet certain requirements.

Set Constructors and Recursive Queries

Fortunately, computing the value of a set constructor is analogous to computing the value of a relation in a database. Thus, there is a fairly substantial body of relevant research. In the context of a database some relations are very easy to compute, namely, those which the database itself is composed of. Other relations may be expressed in terms of those base relations through standard relational operators such as join and select. A more difficult problem is referred to in the database community as computing the results of *recursive queries*, and it is this problem which

relates directly to the problem of computing set constructors such as the one used in the directory tree. Bancilhon and Ramakrishnan [3] provide an overview of several techniques for computing the results of recursive queries. We will look at how these techniques can be applied to RCMS.

In order to understand how these techniques apply to RCMS, we need to establish a framework for describing recursive query evaluations. This framework is given by Bancilhon and Ramakrishnan [3], so only a brief account is provided here. The model for the problem consists of a database of facts, a set of rules for deriving new relations from those facts, and a query which must be evaluated. Facts in the database are represented as predicates. For example, the fact that Earl is a parent of Terry is represented as `Parent(Earl, Terry)`. A rule defining a new relation is simply a predicate definition. So, using Prolog syntax [51], a new relation `Ancestor()` is defined:

```
Ancestor(X, Y) :- Parent(X, Y).
Ancestor(X, Y) :- Parent(Z, Y), Ancestor(X, Z).
```

A query is a partial evaluation of a relation. For example, `Ancestor(?, Terry)` requires computation of all pairs of the relation whose second term is Terry. The query `Ancestor(?,?)` requires evaluation of the entire `Ancestor()` relation.

The first problem in applying any of the recursive evaluation techniques is that the RCMS model is not identical to the above. Fortunately, it is relatively straightforward to translate from the former to the latter. To begin, a hidden predicate needs to be generated for the membership expression of each dynamic set constructor. So,

```
all X : X.instanceOf() == DirNode & Path(root, X)
```

generates temporary predicate T1():

```
T1(X) :- InstanceOf(X, DirNode), Path(root, X).
```

Each dynamic set constructor corresponds to a query. Thus, the set constructor above corresponds to the query `T1(?)`. In Prolog, disjunction is not permitted in the body of predicate definitions. Rather, a given predicate can occur as the head of more than one rule. RCMS predicate definitions must therefore be translated to disjunctive normal form, and each of the conjunctive terms used to define a separate rule. Thus, the RCMS predicate definition:

```
define Path(X, Y) <- X in Y |
   there exists Z [Z in X] & Z.instanceOf() == DirNode & Path(Z, Y);
```

becomes:

```
Path(X, Y) :- In(X, Y).
Path(X, Y) :- In(X, Z), instanceOf(Z, DirNode), Path(Z, Y).
```

Notice as well, that existential quantification in RCMS translates into the use of free variables in the body of the rule. Finally, it should be emphasized that expressing the RCMS model in this manner in no way implies that Prolog should be used to actually implement evaluation of dynamic set constructors. It simply provides a convenient way to show the relationship between evaluation of dynamic set constructors and evaluation of recursive queries.

A more intuitive representation of the predicates and rules involved in a given recursive query evaluation is as a graph. This graph is composed of two kinds of nodes and two kinds of directed edges. The first node type, shown as a rectangle, represents a predicate. The second node type, shown as a rectangle with rounded edges, represents a rule. Edges shown as solid arrows connect a rule node to the predicate node which that rule defines. Edges shown as dashed arrows connect a predicate node to rule nodes in which that predicate is used. Figure 26 shows the graph obtained for the directory tree composite. Predicates which only have outgoing edges are termed base predicates. It is assumed that these predicates can be evaluated directly from the database. In the context of RCMS that means these predicates are, in fact, behaviours which can be invoked. The query predicate has only incoming edges. A mutually recursive set of predi-

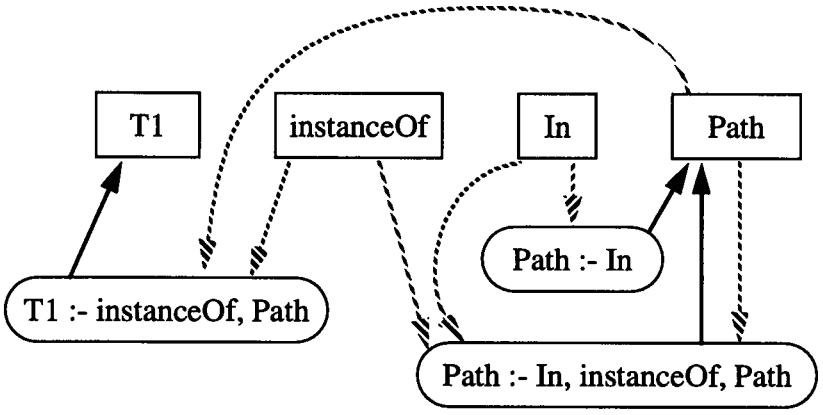

Figure 26 – Predicate/Rule Graph for Directory Tree Composite

cates corresponds to a cycle in the graph. In this example, the predicate `Path()` is trivially recursive. The base cases for the recursion are those rules which are connected to a recursive predicate node, but which are not part of the recursion cycle. A recursive predicate is said to be linear if all of the rules which define it are linear. A rule is linear if it has a single incoming arc which is part of a cycle.

Evaluating Recursive Set Constructors

There are two basic strategies for recursive query evaluation: top-down and bottom-up. A bottom-up strategy starts with the values of the base predicates and constructs the recursively defined relations. Bottom-up strategies compute a large portion of the relation associated with a query. The relevant portions of the relation are then extracted to answer the query. A top-down strategy starts with the query and repeatedly derives portions of the answer to the query. Top-down queries offer the possibility of better performance because they only derive the required portions of the relation. However, the algorithms for top-down evaluation have stronger requirements on the structure of the query. For example, the top-down strategies outlined by Bancilhon

and Ramakrishnan only handle linear recursion. However, they also point out that "there is a belief that most 'real life' recursive rules are indeed linear."

Top-down recursion may also be the most appropriate form for RCMS, due to the nature of the queries associated with dynamic set constructors. These queries all have a single free variable. For example, the query resulting from the set constructor used in the directory tree composite requires evaluation of `Path(root,?)`. Consequently, evaluating a set constructor never requires the full value of a relation to be computed. Nonetheless, because bottom-up techniques have wider applicability, it is worthwhile to consider at least one.

Modified Naive Evaluation

The simplest bottom-up strategy is referred to as naive evaluation. Naive evaluation uses a fixed point iteration to construct the relations corresponding to a mutually recursive group of predicates. The algorithm beings by creating an empty relation for each predicate of the group. These relations are initialized using the non-recursive rules which define the base cases of the recursion. The next step of the algorithm is a loop which computes new values for each of the recursive relations using the current values of these relations and the recursive predicate definitions. The loop terminates when no changes are made to any of the relations being constructed. Figure 27 shows pseudo-code in using the syntax of Raven for naive evaluation of the `Path()` predicate used in the directory tree example. Upon termination of the code segment, the variable `curr` holds a representation of `Path()`. The example assumes the existence of a class `Relation`, which maintains a set of pairs of capabilities, and a class `Pair`. The two elements of a `Pair` object can be accessed as instance variable `X` and `Y` respectively. The example assumes that the `in` operator of RCMS is represented as a base relation referred to by variable `rIn`.

This latter assumption, however, reveals why naive evaluation cannot be directly applied in RCMS. A dynamic set constructor used in the context of a temporal assignment is defined as all

```
var curr : Relation = Relation.new();
var next : Relation;
curr.addRelation(rIn);
while (!curr.equal(next))
    {
    var i, j: Iterator[Object];
    next = curr.copy();
    i = rIn.getIterator();
    while (!i.done())
        {
        var valI : Pair = i.next();
        j = next.getIterator();
        while (!j.done())
            {
            var valP : Pair = j.next();
            if (valI.Y.instanceOf() == DirNode)
               if (valI.Y == valP.X)
                  curr.addPair(valI.X, valP.Y);
            }
        }
    }
```

Figure 27 – Pseudo-code for Naive Evaluation of Path Predicate

elements of the change set of the composite which satisfy the membership expression of the constructor. In order for the routine in Figure 27 to compute the correct value of `Path()`, the value of `rIn` must contain all pairs `(x,y)` such that `x` and `y` are in the change set and `y in x` is true. Because the `in` operator is implemented in terms of iteration, `rIn` must be constructed by computing the change set and then iterating over each element. Since the motivation for using recursive query techniques was to avoid computing the change set, this version of naive evaluation is not useful.

By slightly modifying the naive algorithm, however, it is possible to obtain a technique which has a smaller application domain, but which retains its efficiency advantage. The change consists of computing the base relation on the fly. Rather than assuming that the complete base relation is available from the beginning, we assume that the current values of the base relation can be used to compute further values. This extension of the base relation is performed at the begin-

```
var i : Iterator[Object] = rIn.getIterator();
var new_rIn : Relation = Relation.new();
while (!i.done())
    {
    var val : Pair = i.next();
    var start : val.Y;                                          [a]
    if (start.canUnderstand("getIterator"))
        {
        var j : Iterator[Object]= start.getIterator();
        while (!j.done())
            {
            var end : Object = j.next();
            new_rIn.addPair(start, end);                        [b]
            }
        }
    }
rIn.addRelation(new_rIn);                                       [c]
```

Figure 28 – Incremental Computation of the `rIn` Relation

ning of each iteration during computation of the target (recursive) relation. The new value of the base relation is then used with the base rules and the recursive rules to compute a new value for the target. Figure 28 contains a code fragment for performing incremental evaluation of the `rIn` relation, which corresponds to the `in` operator of RCMS. This computation is possible because the `in` operator maps on to iteration in Raven. For each of the current values of `rIn`, the second element of the pair is extracted (line [a]). By iterating over this value (if possible), new values of `rIn` are generated and added to a temporary relation (line [b]). Finally, once all the current values of `rIn` have been exhausted, the new values are merged with the old (line [c]).

Consider how the modified algorithm works with the `Path()` predicate. Figure 29 shows a small directory tree. The rIn relation is initialized by starting at the root and computing all possible values, resulting, in this case, in the single pair (A,B). The main loop is then entered. First, `rIn` is extended. Pairs (B,C) and (B,D) are added. Next the base rule is applied, generating the values (A,B) (B,C) and (B,D) for `Path`. Applying the recursive rule generates (A,C) and (A,D). This completes the first iteration. Once again, `rIn` is extended, and the only new value is (C,E).

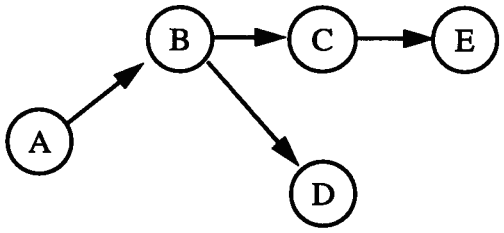

Figure 29 – A Simple Directory Tree

This pair is added to Path, and application of the recursive rules yields (A,E). At this point, `Path` has been completely computed, but the algorithm must make one more iteration in order to detect having achieved the fixed point.

Applicability of Modified Naive Evaluation

The modified algorithm is not as general as naive evaluation. The structure of the predicates used to form the recursive query are restricted. In order to properly describe the nature of these restrictions, the rule graph alone is insufficient. In particular, it is important to distinguish between the bound and free arguments of each predicate used in a rule. We represent this additional information via a superscripted string attached to each predicate, referred to as the *annotation* for that predicate. The string consists solely of a sequence of the letters `b` and `f`. There is one letter for each argument of the predicate, with the first letter corresponding to the first argument, and so on. The letter `b` indicates that its corresponding argument is bound in that use of the predicate. The letter `f` indicates that the argument is free. A free argument indicates that the predicate is being used to query the database. Consider the query which is used to evaluate the set constructor for the directory tree; its annotated version looks like:

```
T1^f(X) :- Path^bf(root,X), instanceOf^bb(X,DirNode).
```

The argument to predicate `T1()` is free because this is the principal query we are attempting to evaluate. Thus, it is marked with a single superscripted `f`. The first argument to `Path()` is a specific object, therefore, it is bound. The second argument is the free variable of the principle query, hence, it is free. Accordingly, `Path()` is marked with a superscripted `bf`. Evaluation of the query represented by this use of `Path()` produces a set of values bound to `X`. As a result, both the arguments to the predicate `instanceOf()` are bound and the predicate is marked with the annotation `bb`. This method of annotating predicates is similar to that used for *magic sets* [3], an optimization technique used in Prolog.

These binding annotations are needed because they tell us when there is a a viable mechanism for evaluating the query corresponding to a predicate. Consider, for example, the built-in operator `in`, represented here as the predicate `In()`. Suppose that it is used with its first argument bound and its second free, that is, it appears in a rule as $\texttt{In}^{\texttt{bf}}\texttt{(X,Y)}$. This represents a query for all objects which can be obtained by iterating over `X`. Clearly, this query can be efficiently computed by doing just that, iterating over `X` and binding the result to `Y`. On the other hand, $\texttt{In}^{\texttt{f-b}}\texttt{(X,Y)}$ represents a query for all objects which produce `Y` when they are iterated over. There is no viable way to evaluate this query in Raven because, in general, objects do not maintain back pointers to the objects they are contained in. That is, the only way to evaluate this query is to examine every object in the system and check to see if it contains `Y`. Clearly, this is not acceptable.

Base predicates with all arguments bound are simple tests on the values of their arguments. Thus, they are straightforward to evaluate. Currently, the only base predicate with free arguments for which bindings may be computed efficiently is $\texttt{In}^{\texttt{bf}}\texttt{()}$. This is not a substantial restriction, however, as the `In()` predicate corresponds to Raven's generic iteration mechanism, which is supported by a wide variety of classes. Furthermore, it is a straightforward task to

extend the system to handle other base predicates. All that is required is that the predicate be associated with an underlying technique for computing the set of values of the free argument given values for the bound arguments.

In order for the modified version of naive evaluation to be correct, all non-recursive predicates used in the rules must be of a form such that it is possible to efficiently compute their bindings. Currently that means non-recursive predicates must have the annotation `bb`, or be a use of the `In()` predicate with annotation `bf`. The motivation for this restriction is clear: in order to incrementally evaluate the base predicates, RCMS requires a means to evaluate the queries corresponding to those predicates.

Top down Evaluation of Recursive Set Constructors

The top-down technique for evaluating recursive set constructors relies on the same annotated predicates as the modified version of naive evaluation. It also places several restrictions on the nature of the recursive query corresponding to the set constructor. First, the query must be linearly recursive. Second, recursive predicates must have a single unbound argument. Third, an unbound argument must appear on the right hand side of the rule in which it occurs. Fourth, all base predicates must either have all their arguments bound or be of a form such that their bindings can be computed efficiently.

Compilation begins by producing an annotated rule set for the query. In the case of the directory tree this rule set is:

```
Path^bf(X,Y) :- In^bf(X,Y).
Path^bf(X,Y) :- In^bf(X,Z), Path^bf(Z,Y).
```

A generator routine for each recursive predicate is created. The arguments to the routine are the bound arguments of the predicate. Each such routine returns the set of values for the unbound

argument which satisfy the predicate. In this example, a generator for $\texttt{Path}^{\texttt{bf}}()$ is created. The Raven behaviour declaration for this generator is:

```
behav Path_bf(X:DirNode) : Set[Object]
```

This declaration states that `Path_bf()` takes a single parameter of type `DirNode` and returns a value of type `Set[Object]`. The body of this routine is compiled from the rules which define the recursive predicate. Each rule generates a portion of the body of the routine. The technique used to compile a rule depends on the structure of the rule.

Rules using only base predicates are compiled left to right. Each base predicate with an unbound argument is converted into an iteration which produces a set of bindings for that argument. Base predicates with no unbound arguments produce tests on values bound via the iterations. If the test fails, then the inner-most iteration containing that test is advanced to its next value. In the body of the innermost iteration, no unbound arguments remain. Any remaining predicates must therefore have their arguments completely bound and can be compiled into simple tests as noted above. If all of these predicates evaluate to *true*, then the value currently bound to the unbound argument of the recursive predicate is added to the result set. Returning to the directory tree composite, the base rule is:

$\texttt{Path}^{\texttt{bf}}\texttt{(X,Y) :- In}^{\texttt{bf}}\texttt{(X,Y).}$

This generates the following iteration:

```
var i : Iterator[Object] = X.getIterator();
while (!i.done())
    {
    var Y : Object = i.next();
    result.add(Y);
    }
```

Compilation of a rule which uses a recursive predicate also occurs left-to-right. Base predicates are handled in the same manner as for non-recursive rules. Evaluation of the recursive predicate is transformed into an iteration. First, a call to the routine corresponding to the recursive predi-

cate is made. The set returned by this invocation is then iterated over to produce bindings for the unbound argument in the predicate evaluation. So, returning to the `Path()` predicate, we have:

```
Path^bf(X,Y) :- In^bf(X,Z), Path^bf(Z,Y).
```

The code fragment corresponding to this rule is shown in Figure 30. In this case, the unbound argument to $\texttt{Path}^{bf}()$ is Y. Base predicate `In()` is compiled into an iteration which binds values to the variable `Z`. This variable is then passed as the argument to `Path_bf()` which returns

```
var t1 : Iterator[Object] = X.getIterator();
while (!t1.done())
    {
    var Z : Object = t1.next();
    var t2 : IdSet[Object] = Path_bf(Z);
    var t3 : Iterator[Object] = t2.getIterator();
    while (!t3.done())
        {
        var Y : Object = t3.next();
        result.add(Y);
        }
    }
```

Figure 30 – Code Fragment for Recursive Rule

a set of values for Y. This set is stored in variable `t2`, and iterated over. Each execution of the body of this inner loop binds a new value to Y. Since there are no further predicates in the rule, the body of the loop simply adds the value of Y to the result set.

Combining the fragment of Figure 30 together with the code derived from the base rule and some headers and trailers, creates a single recursive routine, `Path_bf(X)`, which computes the value of the set constructor `all Y : Path(X, Y)`.

This compilation strategy is similar to backward chaining of the sort typically used to implement Prolog interpreters. However, restrictions on which variables may be unbound allows the resolution phase to be replaced by iteration making it possible to compile the rule set into a relatively efficient set of routines. As with the predicate evaluation mechanism of RCMS, these rou-

tines require augmentation with a memoing facility in order to correctly handle cyclic data structures.

Distributed Evaluation of Recursive Set Constructors

The set evaluators created by the top-down compilation technique work correctly in the distributed Raven environment because Raven provides location transparent invocation. However, if the elements of the set being generated are distributed over a number of sites, evaluation of a set constructor incurs a large communication overhead. Consider for example, a relatively simple situation in which a directory tree is distributed between two sites. The root and some portion of the tree reside at site `A` and a sub-tree resides at site `B`. Call the root of the sub-tree at site `B`, `rootB`, and the root at site `A`, `rootA`. In order to compute the set constructor, RCMS invokes `Path_bf(rootA)`. At some point in the depth first traversal of the tree, a recursive invocation to `Path_bf(rootB)` is made. The body of `Path_bf()` performs two iterations over `rootB` in order to access its children. In each case, obtaining an iterator for `rootB` requires a remote invocation, and each access to a child requires one or more remote invocations. Furthermore, recursive invocations of `Path_bf()` occur for each of the children of `rootB`, thus, requiring even more remote invocations.

This plethora of remote invocations leads to high communication overhead. Evaluation of set constructors is more efficient if it is localized to the sites where the members of the set reside. The functional nature of set constructors makes it possible to achieve this without imposing a large amount of coordination overhead. For the moment, let us restrict our attention to predicates with one bound argument and one unbound argument. The set constructor routines generated for such a predicate take a single argument and that argument is invariably used in combination with a base predicate to generate a set of bindings for a free variable. Thus, when-

ever such a constructor is provided with an argument which resides at a different site, it is worthwhile to move evaluation of the constructor to that site.

In order to achieve this, the generated set constructor behaviours require a slight modification. Upon entering such a behaviour, we check whether the argument is local is not. If it is, then execution of the main body of the behaviour occurs. If it is not, however, we must force evaluation of the behaviour to occur at the remote site. The value returned by this remote evaluation is then returned to the caller.

The behaviours generated for set constructors are part of the placeholder class which RCMS generated when compiling a composite class (ref. Section 5.1). Thus, in order to force invocation of such a behaviour at a remote site, we require an instance of the placeholder class at the remote site. As evaluation of set constructors does not change the state of the placeholder object, it is sufficient to make a temporary copy of placeholder object.

Transfer of the evaluation of a set constructor from one site to another relies on the fact that each site contains a single RCMS system object. This object provides a number of low-level services used by RCMS. Having determined that a particular set constructor has been invoked with a remote argument, we invoke the behaviour `moveEvaluation()` on the local RCMS system object, providing it with the capability for the placeholder object, the name of the predicate being evaluated, and the remote argument. The local RCMS system object uses the remote argument to identify the RCMS system object at the remote site. It invokes the behaviour `receiveEvaluation()` of the remote RCMS system object, using a feature of the Raven system to force a copy of the placeholder object to be created at the remote site, and providing, in addition, the name of the predicate being evaluated and the remote argument. Raven's remote invocation mechanism now moves the computation to the remote site. Behaviour `receiveEvaluation()` is activated at the remote site and invokes the set constructor behaviour on its

local copy of the placeholder object providing the argument received from the other site. Evaluation of the constructor proceeds and eventually returns a value for the set to `receiveEvaluation()`, which, in turn, returns it to the original site.

With these modifications, evaluation of the set constructor now moves from site to site, and expensive iteration operations are performed locally. Two additional modifications further improve the performance of the algorithm. First, the predicates used in set constructors frequently do not reference the state data of the placeholder object at all. This is true, for example, of the `Path()` predicate which is defined solely in terms of its arguments. In this situation, it is unnecessary to generate a copy of the placeholder object at the remote site. Instead, a minimal object is allocated at the remote site, and a low-level Raven primitive is used to force it to become an instance of the placeholder class. This object is internally inconsistent, because it lacks the proper structure of an instance of the placeholder class, but because the state of the placeholder is not accessed by the set constructor behaviour, this inconsistency cannot be observed.

The second modification improves performance when copying of the placeholder object is required. If evaluation of a set constructor moves to and from a given site several times, the original strategy creates copies of the placeholder object each time that evaluation moves to a given site. As the placeholder objects do not change state during evaluation of a set constructor, the temporary copy is cached at each site. The capability of the original placeholder object is passed as an additional parameter to remote sites and used to index the cache. Entries must be flushed from the cache via notification from the original placeholder object that evaluation of the set constructor is complete. In this manner, the evaluation of a set constructor causes the creation of at most one temporary copy of the placeholder object at each site.

6.2 Object Monitoring

One of the more valuable lessons learned from RCMS is that the effectiveness of a configuration management tool is strongly affected by the variety and flexibility of the monitoring techniques available. That this point was not obvious is due, at least in part, to the nature of existing configuration management tools. You do not need monitoring for `make` [18], because it is not an active configuration management tool. The programmer decides when validation/repair occur. In Young's work with the Conic system [58], there is only one type of change in the system which the configuration manager responds to and that is the disappearance of a object due to its failure. With NMS [57], there is an underlying discussion that the status of the network is represented in the database, but virtually no discussion of how that information comes to be there. In implementing RCMS, we came face to face with the need for good monitoring mechanisms. Our initial solutions to the problem are presented in Chapter 4, where it was noted that further development of these techniques would improve RCMS. This section addresses some of the possibilities.

Recall from Chapter 4 that RCMS currently supports three types of monitoring: polling, real time, and error trapping. The most general of these is real-time monitoring, but it is also one for which an object-oriented style of programming poses the greatest difficulties. Almost all instance data in an object consist of references to other objects. When an invocation is made on an object, the value returned from that invocation may be a function not only of the instance data of the object invoked upon, but also of the instance data of any objects it refers to, and the objects they refer to, etc. In general, real time monitoring must detect any change in the abstract state of a component.

For example, Section 5.3 presents a sample composite object which has a container object as one of its components. Each of the items in the container is associated with a cost, and the con-

tainer object supports a behaviour to compute the average cost of the items which it holds. This average cost is part of the abstract state of the composite. For example, the programmer can create a constraint which requires the average cost to be less than some value. What is important to note about this example is that a change in the cost of one of the items in the container, an object which *not* an instance variable of the composite, can create the need to re-validate the composite.

In general, the state of a composite can be affected by any object which is in the change set of the composite. Consequently, RCMS must monitor the state of all objects in the change set of the composite. However, the change set for even a simple composite is very large (ref. Section 5.1.3 and the discussion of iterable membership expressions). Monitoring each of these objects is not acceptable because of the additional overhead it incurs, both as a result of having to manage the monitoring information and because invocations on a monitored object are more expensive.

The following sections present three new monitoring schemes which attempt to address these shortcomings. The first two rely on new instance variable attributes, `partof` and `local`, which provide RCMS with additional semantic information. The third uses the constraints from the composite object definition to create monitoring code which is tailored to the composite.

6.2.1 Part-Of Monitoring

The instance variable attribute `partof` is currently being added to Raven [19]. The semantics of `partof` are quite complex and deal with a variety of issues such as concurrency control, object migration, and persistence. However, intuitively if the programmer uses the `partof` attribute on an instance variable declaration, it signifies a tight binding between the object containing the instance variable and the object referred to by the instance variable. The `partof`

attribute can, therefore, serve as a monitoring hint for RCMS. Rather than monitor all objects in the change set of a composite, RCMS can monitor only those which are bound to it via the attribute `partof`. This style of monitoring works well if programmers use the attribute `partof` to mark instance variables which can affect the abstract state of an object.

An important property of the `partof` attribute is that an object may only be part-of at most one object. When a capability is assigned to an instance variable with the `partof` attribute, the run-time checks whether the assigned object is currently part-of another object. If so, a run-time error is signalled. Otherwise, the assigned object is tagged, and a reference back to the containing object is associated with it. The part-of information stored by the run-time creates a multi-way tree rooted at each object which contains part-of instance variables. Cycles in part-of references are disallowed. This, in conjunction with the restriction that an object may be part-of at most one other object, guarantees that the part-of tree is, indeed, a tree. The presence of back-references in objects which are part-of another object means that it is possible to traverse the tree starting at any node, not just the root.

To use the part-of information, RCMS would emit a modified `updateComponents()` behaviour. Following construction of the components set, the part-of tree for each component would be traversed. Each object in each tree would have monitoring enabled and a flag set indicating that monitoring was induced via the part-of attribute. A back-reference to the composite at the root of the part-of tree would be added to the object. Detection of a change to the object would initiate validation/repair of the corresponding composite.

Upon removal of an object from the components set of a composite, its part-of tree would again have to be traversed in order disable monitoring. Furthermore, changes in the structure of the part-of tree itself would require modification of monitoring status. Fortunately, the support for the `partof` attribute already necessitates routines which update the part-of tree when an

assignment is made to an instance variable with the `partof` attribute. The old value of the instance variable forms the root of a part-of tree which is pruned from the node representing the object whose instance variable was assigned a new value. If monitoring were enabled for this object, then it would be necessary to disable monitoring in all nodes of the pruned sub-tree for which monitoring was inherited. The new value of the instance variable forms the root of a part-of tree which is being grafted on to the node representing the object whose instance variable was assigned a new value. Monitoring status must be propagated to all the nodes of this new sub-tree.

6.2.2 Local Monitoring

Another instance variable attribute which would be useful to RCMS, but which is not currently planned for in the development of Raven, is one indicating that the value of an instance variable is never propagated outside of the object in which it resides. It might, therefore, be called the `local` attribute. An object referred to by an instance variable with the attribute `local` could only be modified by the direct action of a behaviour associated with the owner of that instance variable. Local instance variables would necessarily be declared private as well.

The benefit to RCMS of local instance variables is that they eliminate the possibility of state changes being hidden. In fact, a local instance variable need not be monitored at all. It is sufficient to monitor the object in which it resides. Invocations on local variables would then be able to use the most efficient of Raven's several invocation mechanisms, which is roughly equivalent to a C procedure call. Careful use of the local and partof attributes would allow the construction of objects which RCMS could monitor accurately and with reasonable efficiency.

6.2.3 Specific Monitoring

Another way to increase the monitoring accuracy of RCMS is to use the constraint specification of a composite object to define more precisely the events to be detected. Intuitively, the concept is simple. Suppose, for example, that a composite object has a component `c1`, which has an attribute `up` which takes on the values *true* and *false*. Suppose further, that one of the constraints for this composite is `c1.up`. Every time instance variable `up` of `c1` is modified, its new value is checked. Only if its value is *false* is validation/repair of the composite initiated.

Practical use of this sort of specific monitoring requires that a number of requirements be met. We need to classify what types of constraints permit specific monitoring, to determine how this specific monitoring interacts with the other monitoring techniques of RCMS, and to determine what implementation support is required to provide the service.

Specific monitoring is useful if it allows the more general monitoring techniques to be avoided. Monitoring the value of a given instance variable of a particular object will not provide any improvement if other constraints of the composite require a more general monitoring scheme for that object. To continue with the example from above, suppose that another of the constraints is `c1.utilization() < 50`. In general, it is not possible to determine the specific conditions which result in this constraint failing. To do so would imply that it was possible to analyze the code for the behaviour `utilization()` and determine what it computed, which is just the halting problem in another guise. Thus, the existence of such a constraint forces RCMS to adopt a general monitoring strategy for object `c1` and makes a specific monitoring strategy pointless.

The circumstances in which specific monitoring can be used for a given constraint are partially determined by the basic conditions which the monitoring code can detect. The simplest system would permit comparison of an instance variable against a constant value. A constraint such as

`c1.up` could be handled by this system, but the constraint `c1.up | c1.testMode` could not. The latter constraint fails only when the values of both attributes are false. Detecting failure of this constraint at `c1` requires that the local monitoring code be capable of evaluating a conjunction of conditions. In effect, the local monitoring code must contain an interpreter capable of evaluating logical expressions. The complexity of the expressions which it is capable of handling determines the structure of constraints for which specific monitoring is possible. In order to make presentation of this monitoring scheme more concrete, we assume that the local evaluator can handle expressions whose basic factors are comparisons of an attribute with a constant value and which only make use of disjunction, conjunction, and negation. Such an evaluator could be implemented quite efficiently and still provides a reasonable degree of flexibility.

Applicability of Specific Monitoring

Given a specific evaluator, we can now derive the conditions under which a constraint can use this form of monitoring. The basic task is to take a constraint, and distribute its evaluation over one or more local evaluators such that each local evaluator can determine whether the constraint evaluates to *false*. This is equivalent to evaluating the negation of the constraint and checking that the result is true, which turns out to be a slightly easier form in which to deal with the problem. Assume that the negated constraint has been transformed into disjunctive normal form. This expression evaluates to *true* if any of its terms evaluates to *true*. If each term can be handled by a single local evaluator, then the expression can be distributed. Since each local evaluator has access only to the state of the object to which it is attached, it follows that each term of the negated constraint must access only a single object. For example, the constraint:

```
c1.up ^ d1.up
```

is suitable for local monitoring, because the expression:

```
!c1.up | !d1.up
```

can be decomposed into the condition `!c1.up` which is associated with object `c1` and condition `!d1.up` which is associated with object `d1`.

Extending these conditions to constraints which use quantification uncovers another aspect of specific monitoring in need of discussion. In all cases, the domain of quantification in RCMS is finite. Quantified expressions can, therefore, be re-written as finite length disjunctions or conjunctions. That being the case, the conditions outlined in the previous paragraph would appear sufficient to determine whether specific monitoring could be used for a constraint in which quantification is used. Unfortunately, the domain for a particular quantified expression may not be static. For example, the default domain used in RCMS is the components set of the composite. As noted in Section 6.1, some components sets are static and others are dynamic. Furthermore, the issue of whether the components set is static or dynamic affects even those expressions which are not quantified. Consider a simple expression, such as `c1.up`, which uses instance variable `c1` of the composite. If `c1` appears as the target of a temporal assignment, then the object referred to by `c1` may change over time. Clearly, if the value of `c1` is overwritten, then the object referred to by its previous value should have specific monitoring disabled and the object referred to by the new value should have specific monitoring enabled.

For the sake of simplicity, we assume that specific monitoring is only used when it is possible to statically determine which objects are suitable for specific monitoring. This effectively restricts specific monitoring to composites with static components lists. Constraints which are quantified expressions can use specific monitoring if the domain of quantification is the components set of the composite or is an explicitly defined set whose membership can be determined at the time the composite is created. The latter category consists of instance variables of the composite whose values are established by initial assignments and explicit set constructors whose elements can be statically identified. So for example, the mail transport composite of Section 4.4.2 has

instance variables `t1` and `t2` which hold capabilities for the transport service objects. The values of these two instance variables are established by initial assignment constraints and are not otherwise altered. The constraint:

```
for all X [X in {t1, t2}] X.up
```

is compatible with local monitoring because the domain of quantification is an explicit set constructor composed of statically determined elements. The monitoring condition for this constraint is:

```
!t1.up | !t2.up
```

which meets the criteria outlined above and can clearly be distributed to objects `t1` and `t2` for local evaluation. Unfortunately, one of the constraints of the mail system is actually:

```
there exists [X in {t1, t2}] X.up
```

for which local monitoring is not suitable because the evaluation condition is:

```
!t1.up & !t2.up
```

which cannot be distributed for local evaluation because it requires the state of both `t1` and `t2`. It is possible to construct a more elaborate scheme in which such conjunctive conditions are evaluated by forwarding information from components to an evaluator at the placeholder object, but without first gaining some experience with the simpler system it does not seem reasonable to pursue this.

Implementation Requirements of Specific Monitoring

The changes to the system required to support specific monitoring are quite straightforward. The RCMS compiler would have to detect when specific monitoring was possible. If a composite has a static components list, then the monitoring conditions for each constraint are computed, put in disjunctive normal form, and checked to determine whether local evaluation can be used. The

monitoring condition for each object is determined and recorded. For each object which has a local monitoring condition, RCMS checks the set of constraints to determine whether more general monitoring techniques are required for that object. If so, its local monitoring information is discarded. At the end of the process, if any objects with local monitoring conditions remain, the RCMS compiler generates initialization code for the placeholder object which installs the monitoring condition for each locally monitored object.

Specific monitoring actually occurs in the monitoring post-handler. An object subject to specific monitoring must have associated with it a list of triplets. The first element of the triplet is a list of instance variables being monitored. The second element is the expression to be evaluated and the third element is the capability of the placeholder object. The list permits different composite objects to install different monitoring conditions for the same object. Upon gaining control following an invocation, the post-handler performs the same task for each element of the list. It determines whether control is returning from an instance variable access behaviour corresponding to one of the variables stored in the first element of the triplet. If so, it uses a generic routine to evaluate the condition which is the second element of the triplet. Finally, if the condition evaluates to *true*, the `checkAndRepair()` behaviour is invoked on the placeholder object identified by the third element of the triplet.

6.3 Locking and Composite Objects

One of the assumptions inherent in the RCMS model of configuration management is that there is a consistent state of the composite which is used by RCMS during the validation/repair process. However, obtaining global state information in a distributed system is known to be a complex issue [8] and in some circumstances queries regarding the global state may not even make sense [33]. Much of the work regarding evaluating the global state of a system has assumed a model in which the system may be observed, but not acted upon. This is highly appropriate for

distributed debugging, for example, in which the interference of the debugger may alter the behaviour of the program under observation. If one is willing to allow the observed systems to be disturbed, however, evaluating global predicates becomes an simpler task. Cooper and Marzullo [14], for example, present several algorithms for evaluating global predicates, one of which is "computationally cheap, but may block the monitored program." As RCMS is already invasive, that is, it does not simply monitor objects, but acts upon them, there is less motivation for using non-interfering techniques to support consistent state detection. Therefore, RCMS locks all components before validation in order to create a consistent view of the state of the composite.

Locking in Raven

The proposed mechanism for allowing RCMS to observe a composite object in a consistent state is an extension of the object locking services already provided by Raven. Raven provides fairly standard locking based on shared and exclusive locks. Shared locks allow several different threads to access an object simultaneously, under the assumption that none of those threads is altering the state of the object. An exclusive lock provides mutual exclusion and is used when a thread needs to modify an object. Lock support in Raven consists of a single routine for locking objects:

```
lockObject(objID, lockType)
```

which obtains a lock for the calling thread on the object referred to by `objID`. Parameter `lockType` is either `shared` or `exclusive`. The caller is blocked until the requested lock can be granted. Similarly, there is a single routine for unlocking objects:

```
unlockObject(objID)
```

which removes the calling thread's most recently acquired lock on the object referred to by `objID`. When a thread begins an invocation on an object which has the `controlled` property

it acquires a lock for the object. The Raven compiler assigns the lock type for each behaviour of a class, but these computed lock types may be overridden by using explicit lock type specification keywords or lock control statements. The locks behave in the usual manner, and a given thread may acquire several locks on a given object. This typically occurs when one object makes an invocation on another, which in turns makes an invocation on the original.

RCMS already makes use of the existing locking features of Raven. The placeholder object, as noted in Section 5.2, has the `controlled` property. Certain data structures within the placeholder object, such as the predicate evaluation stack, are not designed for concurrent access. Accordingly, the `checkAndRepair()` behaviour is declared as requiring a write lock. This guarantees mutual exclusion on the placeholder object and preserves the consistency of its data structures. In addition, it effectively serializes all of the validation/repair cycles on a given composite. However, this locking is not sufficient. The validation/repair cycle must have a consistent view of the state of the composite. Locking the components during validation/repair achieves this goal.

Recovering From Deadlock

Intuitively, the easiest way to ensure a consistent state of the composite for the validation/repair cycle, is to obtain exclusive locks for all of its components. These locks would be held until the validation/repair cycle completed. The problem with this simple scheme is that it can result in deadlock. Imagine a composite object with two components C_1 and C_2. One of the behaviours of C_1 invokes a behaviour of C_2. Suppose that a change to C_2 causes RCMS to initiate validation/repair. It begins by acquiring an exclusive lock on C_2. At the same time, another thread makes an invocation on C_1, acquires an exclusive lock for it and subsequently makes an invocation on C_2. Because RCMS holds a lock on C_2, the thread is blocked. Now, however, RCMS tries to acquire an exclusive lock on C_1, and it, too, becomes blocked. Deadlock has occurred.

In order to detect possible deadlock situations arising from RCMS activities, Raven's locking scheme must be modified. Dealing with deadlock is composed of two sub-tasks: first, detecting that deadlock has occurred and, second, taking some action to eliminate the deadlock. Precise detection of deadlock in a distributed environment has been a research issue for a number of years and is generally acknowledged to be a difficult problem [50]. While RCMS could use an existing deadlock detection algorithm, for our purposes a heuristic technique is adequate and considerably easier to implement. Due to the manner in which RCMS uses Raven's locking facilities, the manner in which deadlock is handled is addressed in a novel manner.

When deadlock occurs in a system, the processes involved in the deadlock are all blocked waiting to acquire locks which they can never obtain. Hence, when deadlock occurs, one of the symptoms is that locks are never granted. This symptom can be used as a heuristic for identifying deadlock. That is, if a process remains blocked for more than a certain time interval while trying to acquire a lock, it can simply assume that a deadlock has occurred. Due to the heuristic nature of this algorithm, false deadlocks can be reported. In many systems, false deadlock is very expensive because once the deadlock has been detected, a process is forced to abandon all the locks which it holds and roll-back any changes which it has made to objects for which it held locks. The wholesale abandonment of locks is the only action which is guaranteed to eliminate the deadlock because it eliminates any contention for resources which are the ultimate cause of deadlock.

Fortunately, RCMS uses locking in such a fashion as to minimize the cost of false deadlock detection. Consider, initially, a composite with a static components set. When RCMS determines that such a composite requires validation/repair, it must obtain exclusive access to all of the components. It does this by requesting write locks for all them. However, until such time as RCMS has been granted the locks on all the components, it does not access the components.

Consequently, RCMS can abandon its locks without any need to roll-back changes in the components. The low cost of abandoning locks means that heuristic deadlock detection via time-outs is a viable alternative for RCMS. The simplest deadlock detection scheme for RCMS thus consists of modifying the lock acquisition routines such that each requested lock is associated with a timer. If a timer expires before RCMS has been granted the lock, it releases all locks which have been granted, terminates outstanding requests for locks, and then waits a random amount of time before attempting to acquire the locks again. Although it is theoretically possible for RCMS to be forced to wait indefinitely with this scheme, it is very similar, for example, to the backoff procedure used with Ethernet [35] which works well-enough in practice.

Improving the Performance of Locking

However, we can add yet another heuristic improvement to this scheme. Many situations in which deadlock arises do not require RCMS to release all of its locks. Frequently, the contention will be for a small number of objects and releasing the locks on those objects would be sufficient to eliminate the deadlock. Consider for example, a composite composed of N components. One of these components, let's say C_1, is being modified by a process and hence is locked. Furthermore, that process is about to make an attempt to modify component C_2. At more or less the same time, RCMS determines that the composite needs validation/repair and obtains locks on C_2 through C_N. It blocks, however, when trying to obtain a lock for C_1. Similarly, the process becomes blocked waiting for a lock on C_2. This is a deadlock situation. Eventually, the timer associated with the attempt by RCMS to acquire a lock on C_1 expires. The algorithm proposed initially would have RCMS release all of its locks. However, this is clearly unnecessary as the only object for which there is contention is C_2. If RCMS is aware that this is the only source of contention, then it simply releases its lock on C_2, attempting to re-acquire it after some time-period has elapsed. There is no need to re-do the work of acquiring the locks for C_3 through C_N.

In general, determining those resources for which there is contention is as difficult as precisely detecting deadlock. However, there is another heuristic mechanism available to RCMS. When the timer associated with a lock request expires, RCMS examines all of the locks which it already holds. Of those locks, the only ones which are released are those for which there is another process requesting the lock. The usefulness of this scheme in reducing the amount of work done by RCMS is difficult to judge without measuring the actual levels of contention in a system actively using RCMS.

If the components set of a composite is *not* static, additional complexities are introduced into the locking scheme. For example, if the change set must be computed in order to derive the components set, then a consistent view of the change set is required. The simplest means of achieving this is to lock each object of the change set as it is first encountered by the graph traversal algorithm which is used to compute the change set. Needless to say this would be very expensive.

The requirement to lock all of the objects in the change set reinforces the conclusion that computing the components set of a composite via the change set is not viable in most circumstances. The change set provides a convenient formalism for describing the semantics of RCMS, but actual composites would have to rely on techniques such as those described in Section 6.1 for efficiently computing the components set. Even those techniques, however, would be required to lock the objects encountered during the evaluation of the recursive predicates, to ensure consistent evaluation of the components set.

Effect of Delayed Reply on Locking

Another factor which complicates locking for RCMS is a feature of Raven known as *delayed reply*. A delayed reply occurs when the `leave` statement is executed in the body of a behaviour. The effect of the `leave` statement is to suspend the execution of the thread which is executing the behaviour and temporarily release the lock that it has on the object. Execution of the thread

resumes when another thread executes a `result` statement directed to the suspended thread. The `leave` and `result` statements are useful for implementing objects such as bounded buffers. For example, the behaviour used to retrieve an item from the buffer can execute the leave statement if it finds that the buffer is empty. Subsequently another thread places an item in the buffer and executes a `result` statement. The first thread is awakened and can retrieve the item.

A thread which has been temporarily suspended because it has executed a `leave` statement has not really completed its operation on an object. Only when the corresponding `result` statement is executed is the thread able to complete. Since RCMS desires a consistent and stable state prior to initiating validation/repair, it is undesirable for any of the components to have suspended threads awaiting `result` statements. Achieving this requires that the locks which RCMS requests for a given object *not* be granted if there are any threads temporarily suspended from execution within that object via `leave` statements. The unfortunate consequence of this is that obtaining the lock may be indefinitely delayed. There is no simple solution to this problem. More robust solutions would undoubtedly require that objects using the delayed reply feature be modified to interact directly with RCMS in order to achieve a consistent and stable state.

6.4 Improving Distribution in RCMS

Distribution in RCMS is already superior to systems such as NMS [57] and Young's configuration manager for Conic [58]. There is no centralized management process in RCMS. Configuration information is distributed throughout the system. Each composite has a corresponding placeholder object which has been specifically tailored by RCMS to serve the configuration management needs of that composite. Nonetheless, further distribution of RCMS activities is possible and is advantageous for two reasons. First, further distribution can increase performance by allowing several sub-tasks to operate concurrently and by reducing communications overhead. Second, distribution is essential in providing greater fault tolerance. Distribution for

the latter purpose is more complicated because consistency in the face of failure becomes important. The performance aspects of increased distribution are examined first.

6.4.1 Performance

There are three distinct activities which RCMS performs: monitoring, validation, and repair. Each activity imposes different restrictions on how distribution may be carried out.

Monitoring

Of the three RCMS activities, monitoring is the most highly distributed in the initial implementation. If the monitoring property is used (as opposed to polling or monitoring for communications failures), then the monitoring task is distributed over every component. While this provides the greatest degree of distribution, it does not necessarily provide the best performance, because communication overhead can be high.

In Section 6.2, specific monitoring was described as a technique to achieve finer grain monitoring, that is, a means to reduce the number of validations performed by RCMS by watching for specific changes in objects known to cause a constraint violation. However, local monitoring also results in a reduction of communication overhead. Recall that when a constraint is found to be suitable for local monitoring, the code to evaluate a portion of that constraint is attached to a particular object. This increases the portion of the monitoring task which can be performed locally and reduces the quantity of information that needs to be sent to the placeholder object.

Validation

In the current incarnation of RCMS, the validation process for a single composite is not distributed at all. There is, therefore, a broad spectrum of possibilities for improving the situation. The ease with which validation can be distributed is very much influenced by the structure of the composite. Initially, we restrict our attention to composites with static components sets and

make the assumption that the components do not migrate. We assume that the programmer can supply *location information* (i.e. which components reside on which machines) at compilation time. Even if this were not the case, the same technique would apply, except that a portion of the compilation process would have to be delayed until the composite was instantiated and location information obtained from the system.

Distribution relies on replication of the placeholder object. There is one placeholder object at each site on which the composite has a component. One of the replicated placeholder objects is designated a master and the rest are slaves. The master site is chosen to minimize communication overhead. A placeholder object contains two pieces of state information which must remain consistent amongst the replicas: the components set and the predicate activation stack. Since we have assumed that the components set is static, maintaining its consistency is trivial. The predicate activation stack is used to detect non-terminating recursion while evaluating a constraint. As such, it should actually be attached to the thread evaluating the constraint, rather than to the placeholder object. Thus, the replicas are essentially read-only objects, whose state is initialized when they are created.

Validation itself is composed of two phases: locking of the components, followed by evaluation of the constraints. As the components set is static and the component locations are fixed, each placeholder replica is made responsible for locking the components at its site. Thus, there can be no locking conflict between the replicas.

Localizing Constraint Evaluation

The idea behind distributing evaluation of the constraints is to localize as much of the constraint evaluation process to the sites at which the components reside. Assuming that object location information is available at compile time, this process is not very difficult. Constants are propagated through predicate evaluations, in order to produce expressions consisting solely of primi-

tive operators and constants. This is not possible when an expression makes use of a recursive predicate or a quantifier with an iterable filter expression. In the former situation, the recursion makes it impossible to eliminate predicate evaluations, because replacement of the invocation of the predicate by its body (with parameters appropriately bound) always introduces new predicate evaluations. In the latter situation, the set of elements constituting the domain of evaluation of the quantifier is not known at compile time, hence binding the quantified variable to a set of constant values is not possible.

The primitive expressions are put in disjunctive normal form. Maximal sub-expressions using objects which are at the same site are identified. Each such sub-expression is given a unique identifier and is tagged with the site identifier at which it should be evaluated. Specialized behaviours could be compiled to evaluate each of these sub-expressions each site could be provided with an interpreter capable of evaluating arbitrary expressions. For reasons of simplicity, the latter option is assumed. Expressions not suited to localized evaluation are left to be handled by the master.

Validation begins when the monitoring system signals the master that the composite has changed state. The master signals each slave, and the locking phase begins. Each slave signals the master when it has obtained all its locks or when the locking system has signalled a locking conflict. If the latter occurs, the master signals all slaves to release their locks. Once the master is informed that all locks have been obtained, including those it needed to obtain itself, it signals the slaves to begin constraint evaluation. Each slave evaluates the sub-expressions tagged for its site. When the evaluation of each sub-expression is complete, its result and the unique identifier for that sub-expression is sent to the master. The master receives each of these intermediate results and combines them to generate a final value for the evaluation of each constraint.

Example of Distributed Constraint Evaluation

To get a more concrete feel for how this scheme operates, consider a small example extracted from the mail transport composite of Section 4.4.2. One of the constraints of that system is:

```
Connected(sorter, lowPrio, hold, input) --> ~AvailLow()
```

By expressing the implication operator in terms of the primitive operators and substituting the definition of `AvailLow()`, we obtain:

```
~Connected(sorter, lowPrio, hold, input) |
~(there exists X [X in {t1, t2}] OKForLow(X))
```

Recall, that `t1` and `t2` are the two transport service objects which the mail transport composite may use to deliver mail objects to another site. Because the filter expression for the quantifier specifies a compile-time evaluable domain, the existential quantification may be replaced by a disjunction of terms, each term corresponding to the quantified expression with the variable of quantification bound to one element of the domain. This generates:

```
~Connected(sorter, lowPrio, hold, input) |
~(OKForLow(t1) | OKForLow(T2))
```

Finally, distributing negation across the last term and substituting the body of `OKForLow()` produces:

```
~Connected(sorter, lowPrio, hold, input) |
~(t1.up & t1.cost < 30) & ~(t2.up & t2.cost < 30)
```

This expression consists only of primitive terms and operators and is in disjunctive normal form. Suppose that the transport service objects reside on two different sites and the remaining portions of the composite reside on a third site. Arbitrarily place the master at this third site. When the validation phase begins, the work is divided up, thus:

```
Site 1: ~(t1.up & t1.cost < 30)
Site 2: ~(t2.up & t2.cost < 30)
Site 3: ~Connected(sorter, lowPrio, hold, input)
```

The values for each expression are sent to the master at site 3. Suppose that the values are labelled `S1`, `S2`, and `S3` respectively. Once the master has all three values it computes:

```
S3 | S1 & S2
```

This value, then, is the final result for the evaluation of the constraint.

Certain ordering properties of constraint evaluation may impose additional restrictions on the distributed evaluation of constraints. As presented in Chapter 4, RCMS supports McCarthy evaluation of constraints and also evaluates constraints in the order in which they are presented within the specification clause of the composite definition. The algorithm outlined above clearly violates both of these ordering principles. It is possible to guarantee that these ordering principles are not violated, but it introduces additional overhead and reduces allowable concurrency.

The basic approach remains the same, but the master maintains a partial order describing which sub-expressions must have completed evaluation in order for evaluation of another to begin. Rather than being signalled to evaluate all of their applicable sub-expressions, slaves are told by the master when to evaluate each particular sub-expression. Returning to the example above, McCarthy evaluation demands that the execution order be: `S3`, then `S1`, and finally, `S2`. All concurrency is eliminated, however, the distribution scheme still reduces communication overhead by localizing the computations on `t1` and `t2`. Without further experimentation, it is impossible to know whether it is worthwhile to sacrifice the ordering principles in order to permit greater concurrency.

Handling Recursive Predicates

As was noted previously, constraints using recursive predicates are not handled by this distribution mechanism. Fortunately, there is an alternative for some predicates. In Section 6.1, a modified version of *naive evaluation* [3] was presented as a way to improve the efficiency of

components set computations. Naive evaluation can be used in a similar manner to improve distribution of constraint evaluation. In the current version of RCMS, constraints are evaluated by using backwards chaining, with memo functions used to eliminate non-termination due to cyclic data. Naive evaluation is a forward chaining strategy, which effectively computes a truth table for a predicate. Evaluation of a predicate within the body of a constraint can, thus, be performed by using naive evaluation to produce a truth table, followed by simply looking up the desired value.

Recall, that naive evaluation iteratively constructs the truth table for a given predicate. In a distributed environment, each site can compute a local truth table for the predicate. These local tables are then sent to the master, which merges the table and performs further iterations to produce the final table. Consider the simple example of Figure 31. The edges between the nodes

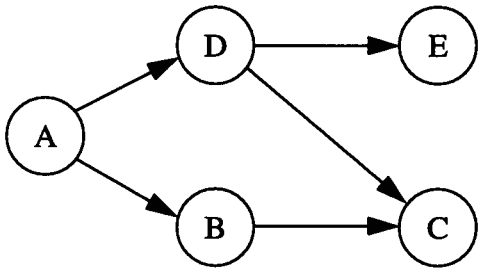

Figure 31 – Distributed Naive Evaluation

represent the predicate corresponding to the `in` operator of RCMS. The goal is to compute the `Path()` predicate, as defined in the directory tree composite. Assume that objects `A`, `B` and `C` are located at one site, and that objects `D` and `E` are located at another. The modified naive evaluation algorithm of Section 6.1 is run at each site. If an edge of the base predicate leaves crosses

from one site to another, then it is included in the table constructed for the base predicate on the source site. The results of this first phase are:

```
Site 1: Path = (A,B)(B,C)(A,D)(A,C)
Site 2: Path = (D,E)(D,C)
```

These values are forwarded to the master, which merges them, iterates, and obtains:

```
Path = (A,B)(B,C)(A,D)(A,C)(D,E)(D,C)(A,E)(A,C)
```

Naive evaluation does not offer much possibility for improved performance because it tends to re-compute large portions of the truth table on each iteration. Much of the work done on other sites is, thus, repeated at the master. However, communications overhead is reduced considerably, because all of the information required from remote sites is obtained at once, rather than through a series of invocations. Furthermore, *semi-naive evaluation* [3] reduces the amount of repeated work in computing a table and could be adapted to RCMS in a manner similar to that used in Section 6.1.

Suppose now that the restriction against objects migrating is removed. How are the distribution techniques outlined above altered? First, migration of a component to a site where there is no replica of its placeholder object requires creation of a new replica. This is not difficult. Also, the locking of a component prior to validation prevents it from migrating once the validation/repair phase has started. However, since objects are no longer fixed, it is of course, not possible to perform the location analysis and *fragmentation* (decomposition of a constraint into sub-expressions that can be executed at one site) at compile time. However, there is no reason that this analysis cannot occur at validation time. A consequence of this is that pre-compilation of behaviours to evaluate particular sub-expressions no longer makes sense, the sub-expressions most appropriate for evaluation at a single site vary as the location of the components changes over time. In this more dynamic context, local evaluation of sub-expressions must be carried out by an interpreter at each site. Since fragmentation occurs at each validation, there is additional

overhead with respect to compile-time fragmentation. Caching of fragmentation results for particular location patterns reduces the penalty incurred.

Locking

The locking phase requires slightly more attention. Each site is supposed to obtain locks only for those objects which are local. The set of replicated placeholder objects must maintain information regarding the location of each component. The system routines responsible for migration can inform RCMS of the movements of objects currently belonging to a composite. This location information might become out-of-date during the locking phase itself, as a component might migrate while RCMS waited to acquire a lock on it. The master placeholder object would not update the location information for the composite until the locking phase was complete in order to avoid having two sites attempting to obtain a lock for the same object. Migrations during the locking phase could, therefore, lead to slaves holding locks on non-local objects. This is a performance issue only, since correct operation of the composite only requires that the locks be obtained and is not concerned with which site locks them. Once locking completes, object location information is updated with any pending migration notifications and fragmentation of the constraints can begin.

Removing the restriction that the components set of the composite be static does not alter the algorithms for locking or constraint evaluation, but does introduce the need for distribution of the components set computations. The only aspect of this task that consumes a significant portion of time is evaluating `indirect` instance variables whose value is specified by a dynamic set constructor. Section 6.1 outlined a mechanism for this in the case of iterable set constructors. Once again, the question of whether the overhead involved in carrying out the distributed computation is offset by efficiency gains, is one which can only be answered through further experimentation with RCMS.

Repair

Given the current implementation of RCMS, improving the performance of the repair phase through better distribution is not practical. The repair process consists solely of executing segments of Raven code. Better performance could only be achieved through automatic extraction of the parallelism inherent in this code. This is a difficult task. The RCMS repair language could be extended to include Raven's parallel programming constructs, and in this manner the programmers could themselves achieve a more efficient repair process.

6.4.2 Fault Tolerance

As was noted at the start of this section, improved efficiency is only one of the reasons for seeking to improve distribution in RCMS. The other major motivation is to achieve a greater degree of fault tolerance. This is a more complicated issue. Even assuming fail-stop behaviour, which is usual for non-critical applications, network partitioning and inconsistency between the various replicas of the placeholder object introduce considerably more difficult problems. Any serious investigation of this topic requires a more thorough investigation into the manner in which Raven handles faults and the way in which this interacts with RCMS. Using distribution to improve fault tolerance in RCMS remains a large and open research issue.

6.5 Better Repair Techniques

The repair facilities offered by the current version of RCMS are the most underdeveloped portion of the system. There is clearly a substantial impedance mismatch between the declarative nature of the specification clause and the imperative nature of the repair clause in a composite definition. It makes it impossible, for example, for RCMS to assess the effects of a repair script until such time as it is executed. It makes the writing and debugging of repair scripts somewhat difficult. It reduces the re-use of repair information, since each programmer is forced to re-solve

simple problems such as re-routing inter-object links. Better repair techniques would make RCMS considerably more pleasant to use.

It is interesting to note that in Young's configuration manager for Conic [58], repair is handled almost entirely automatically. Configuration information in his system consists solely of the structure of the inter-object links of a system. Furthermore, this structure cannot depend on the state of the components. The simple nature of configuration information in his system makes it possible to treat repair actions as a finite search tree. Repair consists simply of searching this tree for the correct configuration given the current structure of the system. Recovery from a failure, however, sometimes requires additional imperative directions from the programmer. Young mentions in passing that an expert system might be used to eliminate this need.

In RCMS, configuration information is represented as a series of logical constraints. Intuition suggests that a constraint solver would be an appropriate repair mechanism. However, because of the general nature of the constraints in RCMS, deriving repair actions frequently requires information outside of the logical domain. For example, a simple constraint such as:

```
foo.bar() > 3
```

cannot be handled by a logical constraint solver, unless the constraint solver possesses a complete semantic description of the `bar()` behaviour of object `foo`. Since formal semantics for the Raven language do not exist, dealing with constraints such as this requires additional direction from the programmer. Using an expert system, this additional information is represented as domain knowledge. It would, perhaps, be possible to use a constraint solver in conjunction with an expert system. The constraint solver would address problems in the logical domain and would invoke the repair system upon encountering constraints outside of that domain. In the interests of reducing complexity, however, an expert system alone appears to be the most practical means for improving repair in RCMS.

Expert systems are a heuristic approach to the repair problem. Formal proof that the expert system can correctly repair an arbitrary invalid composite is not feasible. And without actually having implemented such a repair system, it is difficult to report on the degree to which it is successful. What follows then, is a preliminary proposal for how such a system might be constructed, looking at the interface between the expert system and RCMS and the nature of the rules in the knowledge-base.

The repair phase of the validation/repair cycle occurs when one of the constraints of the specification clause of the composite evaluates to *false*. Clearly, the expert system needs to be provided with the constraint which failed. In order to facilitate manipulation of this constraint, it is represented in disjunctive normal form, with all quantified expressions expanded into finite length conjunctions/disjunctions. This is possible since all quantified expressions in RCMS have a finite domain. The remaining constraints and predicate definitions of the composite are also made available in this form. Finally, access to the state of any of the components is provided.

As noted above, there are two somewhat different tasks which the expert system must perform: solution of a constraint problem at a purely logical level, and application of domain-specific information to handle those aspects of the constraints dependent on the semantics of the component objects being used. In order to perform the first task, predicates and functions which manipulate logical expressions are required, as well as rules describing how constraints can be solved. All of these may be re-used throughout the system.

The state of the expert system is represented as a set of assertions. The two basic assertions are `require(E)`, indicating that the logical expression `E` needs to be satisfied, and `action(S)`, indicating that the statements in `S` need to be executed. The definitions for the constraints and predicates, as well as the states of all the components, are implicitly part of the state of the

expert system. Assuming that the failed constraint is represented by `C`, the expert system begins with the assertion `require(C)`.

The two basic rules used to analyze the constraint describe how to deal with disjunctions and conjunctions. If the constraint which has failed is a non-trivial disjunction of terms, then one of those terms must be made true. The rules:

```
require(E) & disjunction(E) ==> require(left(E))
require(E) & disjunction(E) ==> require(right(E))
```

express this. The predicate `disjunction()` indicates whether its argument is a non-trivial disjunction of terms. One can regard `disjunction(E)` as part of the implicit state of the system, but it is implemented as a function. The functions `left()` and `right()` return the left-hand and right-hand sides, respectively, of their argument. Note that this assumes binary expression trees. One rule is used for each possible manner in which the disjunction may be satisfied. Backtracking ensures that both possibilities are tried if required. The rule for non-trivial conjunctions is:

```
require(E) & conjunction(E) ==> require(left(E)),require(right(E))
```

As expected, the need to satisfy the conjunction is reduced to the task of satisfying both sides. Together, these rules decompose a failed constraint into a set of primitive expressions that need to be satisfied. These primitive expressions consist of relational expressions and evaluation of predicates or negations thereof. Negation can always be distributed across relational operators, hence, negations of relational expressions are not primitives. Non-recursive predicate evaluations can be handled by substituting the body of the predicate with its parameters appropriately bound. If the predicate evaluation is negated, the negation is distributed across the expression resulting from substitution of the body of the predicate.

Recursive predicates require more care. A memoing feature similar to that already used within RCMS to evaluate such predicates is required to avoid non-termination of the substitution procedure. Memoing, combined with the finite domain of a composite, ensures termination. Consider a very simple example using the `Path()` predicate from the directory tree composite of Section 4.4.1. Suppose there is a simple cycle which follows a path from object `A` to object `B` and back again and that the failed constraint is `~Path(A,A)`. Substituting the body of the predicate yields:

```
~(In(A,A) | (In(A,B) & Path(B,A)) | (In(A,A) & Path(A,A)))
```

Memoing is used to prevent further expansion of the `Path(A,A)` term, and distribution of negation gives:

```
~In(A,A) & ~(In(A,B) & Path(B,A))
```

Transformation to disjunctive normal form results in:

```
(~In(A,A) & ~In(A,B)) | (~In(A,A) & ~Path(B,A))
```

The disjunction term on the left is primitive. Assuming that rules have been provided indicating how to satisfy the primitive predicate `In()`, the expert system can satisfy the original constraint by first verifying that there is no link from `A` to itself, followed by breaking the link from `A` to `B`. Note, that if the system had chosen to expand the right side of the expression above, it would have chosen to break the link from `B` to `A` instead.

Handling of relational primitive expressions represents the boundary between the logical domain and the Raven domain. Given a primitive relational expression to satisfy, the system responds first by checking to see whether the expression is currently satisfied. Consider the primitive `~In(A,A)` from the previous paragraph. Re-writing this as `In(A,A) == False` makes it clear that this is, indeed, a primitive relational expression. Assuming that `In(A,A)` is

false, this requirement can simply be dropped. In other words, `require(~In(A,A))` is deleted from the state of the system.

Some domain knowledge about Raven is widely applicable. This is true, for example, of rules concerning the connection and disconnection of inter-object links. Thus, one of the generic rules in knowledge base would be:

```
require(Connected(O1,L1,O2,L2)) ==> action(system.makeLink(O1,L1,O2,L2))
```

Raven also possesses a number of widely used classes for common data structures such as lists, sets, and arrays. As more experience is gained with composite objects using these types of objects, it should be possible to accumulate a generic rule package, thereby greatly reducing the quantity of work required to create composite objects.

The expert system, as described, suffers from the "planning problem." As it decomposes the failed constraint, it accumulates a series of actions which need to be performed. When these actions are actually carried out, the effect of an earlier action may invalidate the reasoning used to derive a later one. This effect stems from the fact that the system does not maintain an internal model of the manner in which actions change the state of the system. One way to solve the problem therefore is to extend the model to provide for representation of the modified state of the composite. However, it is not clear that this additional complexity is worthwhile because there is little data on the complexity of RCMS specifications. Initially, therefore, a more ad hoc solution is appropriate. Responsibility for dealing with the planning problem rests with the programmer. Composite specific rules will embody the knowledge required to avoid difficulties of this sort.

6.6 Using RCMS in a Heterogeneous Environment

As presented thus far in this thesis, RCMS appears to be thoroughly intertwined with the underlying Raven system. Indeed, one of the goals of RCMS is to demonstrate how smoothly config-

uration management can be integrated into an existing system. However, the lessons learned in creating and working with RCMS are much more valuable if they can be easily translated to a more conventional setting.

Currently, in the world of TCP/IP networks, there is a standard for storing configuration information and for performing simple configuration tasks. This standard is the Simple Network Management Protocol (SNMP) [7][46]. In this section, we explore how RCMS can be adapted to work in a heterogeneous environment using SNMP rather than Raven as its foundation.

One of the reasons for working with a system such as Raven is that it provides a very uniform view of the computing environment. All aspects of the environment are represented as objects and all manipulation of the system is carried out via invocations. SNMP, and the related standards for the Management Information Base (MIB) [45] and the Structure of Management Information (SMI) [44], provide a similar type of structure for configuration management in TCP/IP internets, without dictating the use of a particular language or system.

Just as the world-view of Raven is described by a collection of class declarations, so too is the world-view of TCP/IP-based configuration management described by MIB definitions and the SMI. The latter standard simply defines the form of a MIB definition and is not particularly relevant to an examination of erecting RCMS on top of SNMP. A MIB definition, however, describes the set of objects which may be manipulated via SNMP. These objects are simpler than Raven objects, as they consist only of instance variables, which are usually referred to as attributes. Consequently, interaction with the objects is relatively straightforward. SNMP is based on two simple primitives, one to set the value of an attribute and another to fetch the value of an attribute. A third data manipulation primitive allows iteration over tables. SNMP also supports a simple monitoring scheme, called trapping, in which an SNMP object notifies another

process of a major change of state. SNMP-based applications typically use polling to monitor managed objects, while traps are used to signal failures.

A concrete discussion of modifying RCMS to work with SNMP requires a particular target environment. We assume a typical Unix system, with both C and ASN.1 compilers [47]. As SNMP is a network management protocol, we further assume that the system is connected to a network and, obviously, that SNMP is supported.

The RCMS language requires only a small amount of modification. The basic components of composite objects come from the MIB rather than from Raven. In order to access these objects, the language must include an *import statement* which informs the RCMS compiler about MIB definitions which will be referred to in a composite object definition. Given that behaviour invocation and inter-object links are not supported, the interface of a composite consists simply of instance variable declarations. In order to provide reasonable flexibility in defining composites, some of the general purpose Raven classes, such as `IdSet`, continue to be supported. Constraints and predicates retain their existing syntax, with the obvious limitation that behaviour invocation is not supported. As SNMP does not provide support for atomic transactions for managed objects, recoverable repair is not supported. Repair scripts also retain their existing syntax, subject to the obvious limitations.

There are two principle modifications to the compiler. First, in order to support the import statement, it must be capable of parsing and internally representing object definitions from the MIB. As SNMP uses a relatively restricted subset of ASN.1, this does not pose any significant problems. Second, the compiler must generate C rather than Raven. Currently, the output of the compiler is a single Raven class definition for the placeholder object which represents the composite. In the modified compiler, each of the behaviours currently generated for the placeholder object would be translated instead into a C routine. The overall structure of code genera-

tion for these routines remains the same, however, access to the instance variables of the composite and access to managed objects differ. If composite objects themselves are represented in ASN.1, such that they conform to the SMI, then both of these tasks can be handled via SNMP.

Assuming this to be the case, the RCMS compiler creates an ASN.1 definition for the composite. The resulting ASN.1 definition, along with relevant portions of the MIB are passed to an ASN.1 compiler, which generates C data structures corresponding to each object definition and provides C routines for translating to/from these C representations and the corresponding encoded forms. These routines are combined with C routines for performing components set computations, validation, and repair. The resulting C code is compiled and linked with SNMP libraries which provide both the SNMP access routines and a front end which supports those access routines for the composite. This loadable module is used by the RCMS server to instantiate the composite.

Each machine on which composite objects are instantiated requires an RCMS server. It is responsible for initiating validation/repair and for monitoring. Due to the limitations of SNMP, polling is the principle monitoring mechanism. Each composite has a polling period. When this period expires, the RCMS server initiates validation/repair. In order to do this, the code for the composite must be loaded and linked. Depending on the polling period and available resources, this code may be permanently loaded into main memory, or may have to be re-loaded from disk. Similarly, the state of the particular composite may be in memory or may be on disk. Once both the composite and its code are in memory, a new thread is created to carry out the validation/repair. If possible, a generic light-weight threads package would be used.

Although this structure provides some of the key features of RCMS, others are missing. SNMP does not support locking. As noted in Section 6.3, locking is important for obtaining consistent state information from the components. Current applications tend to treat managed objects in

isolation and as a result, consistency is typically not significant. However, as application complexity grows, this issue will assume more importance. Lack of support for inter-object links also limits the utility of RCMS in an SNMP context. Without links, there is no possibility of hierarchically structuring managed objects and the ability to re-use components is low. This deficiency in SNMP stems from its development as a *network* management protocol, rather than a generic management protocol. SNMP is mainly used to instrument network components, and not arbitrary software artifacts. Clearly, in order to provide a general approach to configuration management, which is one of the goals of RCMS, requires a configuration protocol of greater power than SNMP.

CHAPTER 7

Conclusions

The previous chapters of this thesis have introduced, described, and expanded upon a new model of configuration management and have described a new system derived from that model: the Raven Configuration Management System (RCMS). The significant features of this new model of configuration management are that it is *general*, *active*, and *distributed*. The implementation of RCMS has proven the viability of the model. Furthermore, the research has produced a number of insights into the sorts of services which the system must supply in order to make configuration management effective.

7.1 Contributions

There are three distinct areas in which this thesis research has produced significant research contributions. First, *a new model of configuration management* was defined that is suitable for analyzing existing configuration management systems and which provides guidance in the creation of new systems. Second, the *RCMS configuration language*, based on that model, was designed

with emphasis on the general, active, and distributed features. Third, the *RCMS language was implemented.* As a result of this implementation, the sorts of system level services which need to be supplied to a configuration management system were investigated. Additionally, exploration of how to provide those services in an efficient manner was carried out.

The following sections of this chapter provide more detail about the nature of the research contributions made in each of these areas.

7.1.1 A new Model of Configuration Management

Until recently, configuration management has been handled in an *ad hoc* manner. Particular problems begat particular solutions. At a time when computers were few and the degree of connectivity was low, the complexity of the problems posed no difficulties for such an approach. However, as computing power increases while cost decreases and as communications bandwidth increases, there are more systems, more interconnections, and more complex management problems. It is no longer sufficient to simply identify a particular problem and implement a system which addresses it. A general model of configuration management is required. Chapters 2 presents just such a model of configuration management. The important features of this model are that it is general, active, and distributed.

General

The RCMS model of configuration management is based on the insight that all managed entities can be viewed as objects and all management activities as the process of maintaining relationships between these objects. Those objects might be parts of a distributed application. They might be elements of a software development project: design specifications, source code, and documentation. Indeed, they could be any collection of objects related to each other in any man-

ner. However, by adopting a uniform point of view the RCMS model is able to address all of these types configuration management tasks. Thus, the RCMS model is general.

Active

As noted in Chapter 3, most existing configuration management tools provide means for describing configuration information, but few make use of this information to automatically ensure that the system remains in an acceptable state. The RCMS model emphasizes the importance of both monitoring and repair. Monitoring, that is, observing the system in real time, allows problems to be detected. Having repair information as an integral part of the model makes it possible for such problems to be dealt with automatically by the computer system. Thus, the RCMS model is active.

Distributed

The need for configuration management services is acute in distributed systems. For a model of configuration management to be practical it must avoid requiring centralized services which would inhibit its use in a distributed environment. By introducing the concept of the *composite object*, which tightly binds together the objects to be managed, the configuration management information, and the services related to those objects, the RCMS model provides an ideal mechanism for distributing configuration management activities. Thus, the RCMS model is distributed.

7.1.2 The RCMS Configuration Language

The second major contribution of this thesis research is a new configuration language whose design was guided by the RCMS model of configuration management. A concise description of this language is contained in Appendix 1, while Chapter 4 presents the language in a more informal manner through a series of examples. The significant features of this new configuration lan-

guage are that it provides an object-oriented approach to configuration management, a flexible mechanism for identifying objects which are to be managed, and a constraint-based language for describing relationships between objects. In addition, the language is coupled with specific mechanisms for monitoring and repairing configurations.

Object-Oriented

The RCMS model introduces the composite object as a means to group together objects which are managed as a unit. The RCMS configuration language takes the concept of the composite object and makes it concrete. An object-oriented approach to configuration management offers several benefits. Object-oriented programming is familiar to most programmers. RCMS is thus able to present configuration management in a familiar manner. In addition, in an object-oriented system such as Raven, the configuration manager can be very cleanly integrated with the underlying system. Finally, the well-defined interfaces and encapsulation resulting from an object-oriented approach make it possible to decompose hierarchically management tasks into smaller and simpler units.

Identifying Objects to be Managed

An *active* configuration management system responds to ongoing changes. An important aspect of responding to changes is to be able to create configurations whose set of components change over time. For example, if one is managing a set of objects which have a ring-like communication structure, then the configuration containing those objects should respond automatically to the addition or deletion of members from the ring. One of the important features of RCMS is that its mechanism for specifying the components of a configuration is very flexible. RCMS uses logical expressions to specify which objects should be part of a configuration. These logical expressions are re-evaluated whenever the system changes. Consequently, RCMS can respond to changes in the system in precisely the desired manner. The provision of this flexible language

mechanism requires special support in the RCMS compiler which is discussed in Section 5.1.3 and Section 6.1.

Managing via Constraints

Most current configuration management systems adopt an operational point of view. Configuration management is specified in terms of what tasks need to be performed. RCMS adopts a different point of view, namely, that configuration management should be specified in terms of how objects are related to one another. Predicates and logical constraints are the means by which these relationships are established in RCMS. Representing configuration declaratively has several advantages. The notation is concise and intuitive. It is amenable to formal analysis and manipulation. Finally, it permits a flexible repair sub-system because it specifies the ends and not the means of configuration management.

Monitoring and Repair

The RCMS model of configuration management demands that a CMS provide means for monitoring and repairing configurations. Another contribution of this research is an understanding of the importance of the monitoring process to a CMS and the discovery that many sorts of monitoring are possible and desirable. There is no one right way to perform monitoring. RCMS, for example, provides monitoring based on polling, call-backs, and error trapping. The use of these different monitoring techniques is shown in Chapter 4. The examples in Chapter 4 also demonstrate the usefulness of providing both roll-forward and roll-back repair mechanisms.

7.1.3 Implementation of RCMS

The third and final area in which significant research contributions have been made is in providing an implementation of RCMS. First, this implementation establishes the viability of both the RCMS model and the configuration language. Second, the implementation establishes the sort

of services required in order to implement a general, active, and distributed CMS. Finally, several interesting implementation techniques were used or proposed in order to address issues such as efficient evaluation of quantified expressions, preserving consistency through locking, improving the performance of component set computations, and improving the level of distribution of management activities.

Services

As noted in Chapter 5 and Chapter 6 the successful implementation of a CMS relies upon the existence of particular system services. The services identified as crucial during the implementation of RCMS are: monitoring, locking, and error handling. If these services are not provided in a sufficiently powerful and flexible manner, the capabilities of the CMS are reduced.

Efficient Evaluation of Expressions

During the development of the examples presented in Chapter 4 of this thesis it became obvious that efficient evaluation of quantified logical expressions was crucial to the performance of RCMS. A naive implementation, using the change set of the composite as the domain of quantification, is unacceptable. Section 5.1.3 presents an optimization which significantly improves performance in evaluating these types of expressions.

Locking

The actions of the configuration management system must not interfere with the normal operations of the managed objects. RCMS uses locking to ensure that management activities are isolated from the normal activities of managed objects. However, the introduction of additional locking can create deadlock. Section 6.3 describes how RCMS deals with the possibility of deadlock. In particular, the manner in which RCMS acquires and uses locks makes it possible to break deadlocks without requiring management actions to be rolled-back.

Components Set Computations

As with evaluating quantified expressions, the naive method of implementing the powerful mechanism which RCMS uses for specifying the composition of a configuration does not offer acceptable performance. Section 6.1 establishes a connection between this task and the task of evaluating recursive queries in a database. The database results are not immediately applicable, however, necessitating modification of existing algorithms so that they can be used in the context of RCMS. Furthermore, these algorithms are applicable to any CMS which provides object identification mechanism like those of RCMS.

Distribution

Composite objects provide for better distribution of configuration management activities than any existing configuration system. Nonetheless, there is great room for improving the degree of distribution in RCMS, creating both better performance and better fault tolerance. Section 6.4 describes ways to improve distribution of several aspects of RMCS, including computing the components set of a composite, carrying out monitoring, and performing repair.

Implementation of Raven

A final area of research contributions were made with respect to the implementation of the Raven system itself. As has been noted elsewhere, a powerful configuration manager relies on the system to provide services such as monitoring, locking, and error handling. The existing Raven system provided locking mechanisms, but monitoring and error handling facilities had to be added. In addition, because the RCMS compiler translates composite definitions to Raven code, a stable Raven compiler was a necessity. Much work was done on the Raven compiler in order to achieve this goal.

7.2 Validation of RCMS

In addition to considering the research contributions of this work, it is important to evaluate whether RCMS is a useful configuration management tool. Section 2.2 presented a set of nine criteria, representing fundamentally important aspects of any configuration management system. Existing configuration management systems, several of which were examined in Chapter 3, fail to meet these requirements in full. RCMS addresses each of the issues:

(1) Identification: Composite objects themselves are the identification mechanism of RCMS. The use of indirect instance variables, and dynamic set constructors provides a particularly flexible means for specifying the components of a composite object.

(2) Control: Complete integration with the underlying Raven system means that RCMS can access and modify any of the components of a composite.

(3) Specification: The specification clause of an RCMS composite class definition provides the programmer with the power of first order predicate logic to express the required relationships between the components of a composite.

(4) Monitoring: Several different techniques are provided by RCMS for monitoring the state of a composite object. The simplest of these techniques simply allows the state of the composite to be periodically checked. Considerably more complex techniques, allowing monitoring to be distributed to each component of the composite, are described in Section 6.2.

(5) Repair: RCMS offers both roll-back and roll-forward repair. Roll-back repair relies on support for transactions. Any transaction which causes a composite to violate a constraint is aborted, and the composite returned to its previous state. Roll-forward repair attempts to modify the composite to remove the cause of a violated constraint. Currently, roll-forward repair relies on programmer-supplied repair scripts; however, Section 6.5 outlines how an expert system might automate this process.

(6) Non-interference: The locking scheme of Section 6.3 has been designed to provide RCMS with a consistent view of a composite and to minimize the disruption of normal system operation caused by configuration management activities. The pre-emptable locks used by RCMS allow normal system threads to complete their activities within composite objects. The deadlock detection scheme forces RCMS to wait when its activities might lead to deadlock.

(7) Integration: RCMS is cleanly integrated with the underlying Raven system. Raven objects can be used freely within RCMS composites, and RCMS composites can be used freely within the Raven system. The syntax and semantics of composite objects have been kept as close to that of Raven objects as possible. Raven objects typically require no modification to be used as part of a composite.

(8) Performance: Performance issues have been addressed in many ways within RCMS. An extension of Raven's object invocation mechanism allows object monitoring to occur with very little overhead. The logic constraints of each composite are compiled to enhance the speed of validation. Section 6.1 presented two algorithms for reasonably efficient handling of complex composite membership specification. Section 6.4 examined how further distribution of the validation process could improve its efficiency.

(9) Distribution: RCMS associates configuration management information directly with the managed objects themselves. In RCMS, there is no centralized repository of configuration information, and no centralized server responsible for carrying out configuration management tasks. Each composite object is self-contained.

7.3 Future Work

Limitations in the current version of RCMS suggest interesting research issues. The three issues which appear to be most productive to pursue are:

(1) Extended Logic: The logic used to describe constraints in RCMS has no means for expressing temporal relationships. As noted in Section 3.3.3,

temporal relationships are used heavily in the Network Management System, and would very likely, therefore, be useful in RCMS. An obvious extension to RCMS would be to extend RCMS based on an existing temporal logic.

(2) Improved Repair: The repair mechanism in RCMS is primitive. A brief outline of how an expert system might be used to perform repair was given in Section 6.5, however, much work remains to be done on how the verification phase passes information on the nature of the failure to the repair phase. Furthermore, the addition of temporal constructs to the constraint language undoubtedly has interesting ramifications for repair.

(3) Better Fault Tolerance: The current model of configuration management in RCMS essentially ignores the issue of failures. In order to more reasonably support configuration management in a distributed system, failures need to be explicitly accounted for, specifically with respect to network partitions and the effects this has on composite objects which are distributed on both sides of the partition. Another avenue of research is an examination of the exception handling features needed in the underlying system and how they interact with the configuration management system.

7.4 In Closing

The development of RCMS has been very encouraging. Issues of general importance for configuration management, such as locking and object monitoring, have been explored, producing new algorithms that address these issues. More importantly, however, RCMS demonstrates the feasibility and value of general-purpose configuration management in an object-oriented environment. The use of a declarative language for expressing configuration information is both intuitive and concise. The implementation of the system supports a wide variety of optimizations, yielding performance improvements which will allow its practical application. General, active, and distributed configuration management is both desirable and practical.

Bibliography

[1] Eric Allman. *An Introduction to the Source Code Control System.* 1980.

[2] J.S. Auerbach, D.F. Bacon, A.P. Goldberg, G.S. Goldszmidtz, M.T. Kennedy, A.R. Lowry, J.R. Russel, W. Silverman, R.E. Strom, D.M. Yellin, S.A. Yemini. High-Level Language Support for Programming Distributed Systems. *Proceedings of the 1991 CAS Conference*, pages 173-195, 1991.

[3] Francois Bancilhon and Raghu Ramakrishnan. An Amateur's Introduction to Recursive Query Processing Strategies. *Readings in Database Management Systems*, pages 507-548, 1988.

[4] Mario R. Barbacci and Jeannette M. Wing. A Language for Distributed Applications. *International Conference on Computer Languages*, March 12-15 1990, New Orleans, Louisiana, pages 59-68, 1990.

[5] A. P. Black. Supporting Distributed Applications: Experience with Eden. *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 181-193, 1985.

[6] Jeffrey D. Case, James R. Davin, Mark S. Fedor, and Martin L. Schoffstall. Network Management and the Design of SNMP. *Connexions*, 3(3), pages 22-26, 1989.

[7] Jeffrey D. Case, Mark S. Fedor, Martin L. Schoffstall and James R. Davin. *A Simple Network Management Protocol.* Request for Comments 1028, DDN Network Information Center, SRI International, November 1987.

[8] K. M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1), pages 63-75, 1985.

[9] Kai-Hsiung Chang and William G. Wee. A Planning Model with Problem Analysis and Operator Hierarchy. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 10(5), pages 672-675, 1988.

[10] David Chapman. Planning for Conjunctive Goals. *Artificial Intelligence*, Volume 32, pages 333-377, 1987.

[11] Geoffrey M. Clemm. The Workshop System: A Practical Knowledge-Based Software Environment. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, SIGPLAN Notes, 24(2), 1988.

[12] Terry Coatta. Using Objects to Distribute Configuration Management Tasks, *Proceedings of the 1993 IBM CAS Conference*, Toronto 1993, pages 541-552.

[13] D. Cohen. Compiling Complex Database Transition Triggers. *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 225-234, 1989.

[14] Robert Cooper and Keith Marzullo. Consistent Detection of Global Predicates. *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging. ACM SigPlan Notices*, 26(12), pages 167-174, December 1991.

[15] Lois M. L. Delcambre and James N. Etheredge. A Self-Controlling Interpreter for the Relational Production Language. *ACM SigMOD*, 17(3), pages 396-403, 1988.

[16] F. DeRemer and H H. Kron. Programming in the Large versus Programming in the Small. *IEEE Transactions on Software Engineering*, (2)2, pages 80-86, 1976.

[17] J. Estublier. Configuration Management: The Notion and the Tools. *Proceedings of the International Workshop on Software Version and Configuration Control*, German ACM Chapter. Grassau, Germany, pages 38-61, 1988.

[18] S. I. Feldman. MAKE —A Program for Maintaining Computer Programs. *Software Practice and Experience*, 9(4), 1979.

[19] David Finklestien. Object Properties in Raven. Master's Thesis, University of British Columbia, 1994 (to appear).

[20] German Goldszmidt, Yechiam Yemini, and Shaula Yemini. Network Management by Delegation — The MAD. *Proceedings of the 1991 CAS Conference*, pages 347-359, 1991.

[21] William Harrison, Harold Ossher, Mansour Kavianpour. Integrating Coarse-Grained and Fine-Grained Tool Integration. *Proceedings of the 5th International Workshop on Computer-Aided Software Engineering*, pages 23-35, 1992.

[22] Karen E. Huff, Victor R. Lesser. A Plan-Based Intelligent Assistant That Supports the Software Development Process. *ACM SigSoft*, 13(5), pages 97-106, 1988.

[23] John Hughes. Lazy Memo-functions. *Lecture Notes in Computer Science - Functional Programming Languages and Computer Architecture*. Volume 201, pages 129-146, Nancy, France, September 1985.

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

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

[26] G. E. Kaiser and P. H. Feiler. An Architecture for Intelligent Assistance in Software Development. *Proceedings of the 9th International Conference on Software Engineering*, pages 180-188, 1987.

[27] Robert M. Keller and M. Ronan Sleep. Applicative Caching. *ACM Transactions on Programming Languages and Systems*, 8(1), pages 89-108, 1986.

[28] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, Inc. Englewood Cliffs, New Jersey, 1978.

[29] G. Kiernan, C. de Maindreville, and E. Simon. Making Deductive Database a Practical Technology: A Step Forward. *ACM SigMOD*, 19(2), pages 237-246, 1990.

[30] J. Kramer and J. Magee. Dynamic Configuration for Distributed Systems. *IEEE Transactions on Software Engineering*, 11(4), 1985.

[31] J. Kramer and J. Magee A Model for Change Management. *IEEE Transactions on Software Engineering*, pages 286-295, 1988.

[32] J. Kramer, J. Magee, and M. Sloman. The CONIC Toolkit for Building Distributed Systems. *IEE Proceedings-D Control Theory and Applications*, 134(1), pages 73-82, 1987.

[33] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7), pages 558-565, July 1978.

[34] J. Magee, J. Kramer, and M. Sloman. Constructing Distributed Systems in CONIC. *IEEE Transactions on Software Engineering*, 16(6), 1989.

[35] R.M. Metcalfe and D. R. Boggs. Ethernet: Distributed Packet Switching for Local Computer Networks. *Communications of the ACM*, July 1976, pages 395-404.

[36] Naftaly H. Minsky. Controlling the Evolution of Large Scale Software Systems. *Conference on Software Maintenance*, pages 50-58, 1985.

[37] Naftaly H. Minsky. The Imposition of Protocols Over Open Distributed Systems. *IEEE Transactions on Software Engineering*, 17(2), pages 183-195, 1991.

[38] Naftaly H. Minsky and David Rozenshtein. Configuration Management by Consensus: An Application of Law–Governed Systems. *Proceedings of the Fourth ACM SigSoft Symposium on Software Development Environments (SigSoft Software Engineering Notes)*, 15(6), pages 44-55, 1990.

[39] Gerald Neufeld, Don Acton, and Terry Coatta. *The Raven System.* University of British Columbia, 1992. Technical Report.

[40] James L. Peterson and Abraham Silberschatz. *Operating Systems Concepts.* Addison Wesley Publishing Company, 1985.

[41] Erhard Ploedereder and Adel Fergany. The Data Model of the Configuration Management Assistant. *ACM SigSoft*, (17)7, pages 5-14, 1989.

[42] Tore Risch. Monitoring Database Objects. *Proceedings of the Fifteenth International Conference on Very Large Databases*, pages 445-453, 1989.

[43] Thomas Rodden, Pete Sawyer, and Ian Sommerville. Interacting with an Active, Integrated Environment. *ACM SigSoft*, 17(7), pages 76-84, 1988.

[44] Marshall T. Rose and Keith McCloghrie. *Structure and Identification of Management Information for TCP/IP based internets.* Request for Comments 1155, DDN Network Information Center, SRI International, May 1990.

[45] Marshall T. Rose (editor). *Management Information Base Network Management of TCP/IP based internets: MIB-II.* Request for Comments 1158, DDN Network Information Center, SRI International, May 1990.

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

[47] Mike Sample. *How Fast Can ASN.1 Encoding Rules Go.* Master's Thesis, Department of Computer Science, University of British Columbia, 1992.

[48] Eric Emerson Schmidt. *Controlling Large Software Development in a Distributed System.* Technical Report CSL-82-7, Xerox Corporation, 1982.

[49] Michael L. Scott. Language Support for Loosely Coupled Distributed Programs. *IEEE Transactions on Software Engineering*, 13(1), pages 88-103, 1987.

[50] A. Silberschatz, J. Peterson, and P. Galvin. *Operating System Concepts (Third Edition).* Addison-Wesley Publishing Company, Reading Massachusetts, 1991.

[51] Leon Sterling and Ehud Shapiro. *The Art of Prolog.* The MIT Press, Cambridge, Massachusetts, 1986.

[52] Richard N. Taylor, Frank C. Belz, Lori A. Clarke, Leon Osterweil, Richard W. Selby, Jack C. Wileden, Alexander L. Wolf, and Michael Young. Foundations for the Arcadia Environment Architecture. ACM SigSoft, 13(5), pages 1-12, 1988.

[53] Walter Tichy. *An Introduction to the Revision Control System.* Department of Computer Science, Purdue University. 1982.

[54] Ronnie Thompson and Ian Sommerville. Configuration Management Using SysL. *ACM SigSoft,* 17(7), pages 106-109, 1989.

[55] Bernhard Westfechtel. Revision Control in an Integrated Software Development Environment. *ACM SigSoft*, pages 96-105, 1989.

[56] David E. Wilkins. Domain-Independent Planning: Representation and Plan Generation. *Artificial Intelligence*, Volume 22, pages 269-301, 1984.

[57] Ouri Wolfson, Soumitra Sengupta, and Yechiam Yemini. Managing Communication Networks by Monitoring Databases. *IEEE Transactions on Software Engineering*, 17(9), pages 944-953, 1991.

[58] Andrew Jeremy Young. *A Structural Approach to Fault and Change Management in Distributed Systems.* Ph.D. Thesis, University of London Imperial College of Science, Technology and Medicine, 1991.

[59] Moshe M. Zloof. QBE/OBE: A Language for Office and Business Automation. *IEEE Computer,* 14(5), pages 13-22, 1981.

Glossary

before-image - A copy of the instance variables of an object made prior to the execution of a behaviour on that object. The before-image is used to support object recoverability in Raven. An object made be restored to a former state by copying a before-image of that state into the object's instance variables.

capability - A reference to an object in Raven. All objects are accessed via capabilities using Raven's invocation mechanism.

change set - Intuitively, the change set of an object consists of all objects which could have an effect on the state of the object. That is, if a change to an object `X` results in an observable difference in the behaviour of a given object, then `X` is a member of the change set of that object. Formally, the change set of a particular object `X` is defined recursively as the union of the set of objects whose capabilities are stored in the instance variables of `X` and the change sets of these same objects.

check set - A set of events for which the configuration management system (CMS) monitors. When an event from the check set occurs, the CMS checks to see whether the configuration still satisfies its specification.

component - An object which is part of a configuration. In RCMS, a configuration is also called a composite object. The components of a composite object are specified by the instance variables of the composite. The components consist of the union those objects referred to by instance variables without the `indirect` attribute and the contents of the objects referred to by instance variables with the `indirect` attribute.

composite class - The unit of definition in RCMS. A composite class specifies the interface and implementation for those composite objects which are its instances. A composite class definition consists of an interface declaration (which is very similar to a Raven class interface declaration), a specifications clause, which defines predicates, relations, and constraints, and a repair clause, which

composite object - The unit of allocation in RCMS. A composite object is governed by the constraints in the specifications clause of its corresponding class definition. The actions taken by RCMS to maintain the composite in an acceptable state are provided in the repair clause of the class definition.

configuration - A collection of objects which are related to one another and for which there are criteria to establish whether that collection of objects is functioning correctly. In RCMS, a configuration is called a composite object.

configuration management - The process of identifying the relationships which govern the behaviour of a system, recognizing systems in which those relationships do not produce the desired behaviour, and taking corrective action to remedy the problems.

constraint - A logical expression which is part of the specification clause of a composite class definition. The constraints of a composite object are evaluated to determine whether that composite is in an acceptable state. If all constraints of a composite evaluate to *true*, the composite is said to be valid. If any of the constraints evaluate to *false*, the composite is said to be invalid.

deep equal - Two objects are deep equal only if they are members of the same class, and the values for each of their instance variables are also deep equal. Intuitively, deep equality means that two objects represent the same value. For example, one may have two copies of the string "hello world" which reside at distinct locations in memory, but nonetheless they represent the same set of words.

domain of quantification - The set of objects used when evaluating a quantified expression. All quantified expressions in RCMS are evaluated with respect to a finite domain, and hence, are equivalent to finite length conjunctions (in the case of universal quantification) or finite length disjunctions (in the case of existential quantification).

elementary class - The unit of definition in Raven. An elementary class specifies the interface and implementation for those elementary objects which are its instances. An elementary class definition consists of an interface declaration, followed by a series of behaviour implementations.

elementary object - An object programmed using the Raven language. (compare with a composite object, which is one created using RCMS.)

encapsulation - A property of object-oriented systems which refers to the fact that the internal structure of an object is not visible. The behaviour of an object is thus defined solely by the manner in which it interacts with other objects. Objects which exhibit the same behaviour may have implementations which differ significantly and encapsulation hides this fact.

error trapping - One of the monitoring schemes used by RCMS. With this scheme, validation/repair of the composite are triggered by the failure of an attempt to send data over an inter-object link. Such failures occur when the link spans machines boundaries and either the communications system or the remote machine fails.

filter expression - An expression used in conjunction with a quantifier to restrict the domain of quantification. The filter expression should have the quantified variable as a free variable. The domain of quantification is restricted to those objects for which the filter expression evaluates to *true*.

interface declaration - The portion of a composite class definition which describes the external interface of instances of that class. Additionally, the manner in which the instance variables are declared determines which objects are considered components of the composite.

iterable - A property of the membership predicates used with dynamic set constructors. An iterable membership predicate can be compiled into a reasonably efficient routine for computing the set which it specifies. The formal definition of iterable requires that the membership predicate be in disjunctive normal form. A logical expression in disjunctive normal form is iterable with respect to an unbound variable `X` if each of its conjunctive clauses is iterable with respect to `X`. A conjunctive clause is iterable with respect to `X` if at least one of its factors is a non-negated use of the `in` operator with the variable `X` on its left hand side.

iteration - The process of obtaining the contents of an aggregate object, one object at a time. Iteration is the principle mechanism used in Raven for examining the contents of data structures such as lists, sets, and arrays.

object - An entity within a computer system that can be referred to, contains information, and is associated with a set of operations for accessing and modifying that information.

placeholder object - The elementary object which represents a composite object at run time. The RCMS compiler produces one placeholder class definition for each composite class which is defined.

polling - One of the monitoring schemes used by RCMS. With this scheme validation/ repair of the composite occurs at fixed time intervals.

post-handler - A segment of code executed during the invocation process following the execution of the body of the invoked behaviour (see the entry on pre-handlers for further information).

pre-handler - A segment of code executed during the invocation process prior to the execution of the body of the invoked behaviour. Pre-handlers are hooks which provide access to the invocation process to other parts of the Raven run time. For example, recoverability is implemented using a pre-handler which takes a snapshot of an object's instance variables prior to executing a behaviour on that object. This snapshot is used to restore the instance variable's to their original values when needed.

real time - One of the monitoring schemes used by RCMS. With this scheme validation/ repair of the composite are triggered by a change to any of its components. The use of the term "real time" does not indicate that RCMS is capable of hard real time programming involving deadlines and schedules. Rather, it means that the changes to the composite are detected as soon as they occur.

repair clause - That portion of a composite class definition which describes the manner in which RCMS is to modify a composite object in order to return it to a valid state.

roll-forward repair - A repair strategy in which RCMS attempts to return a composite to a valid state by executing repair scripts based on which constraint has failed. The repair scripts are provided as part of the composite definition.

roll-back repair - A repair strategy in which RCMS returns the composite to a valid state by undoing the effects of the invocations which resulted in the violation of one or more of the composite's constraints.

specification clause - That portion of a composite class definition in which predicates, relations, and constraints are defined.

undetected modification - A modification to an object which changes the state of a composite object, but which does not cause the composite to be re-validated. The lack of re-validation means that it is possible for violation of the composite's constraints to go undetected by RCMS.

valid - A composite object is said to be valid if each of the constraints of its specification clause evaluates to *true*.

validation - The process of evaluating all of the constraints of a composite to determine whether it is valid.

validation/repair phase - The process of repeatedly validating and repairing a composite until such time as it becomes valid.

Appendix 1

The Raven Configuration Management Language

A-1.0 Introduction

The Raven Configuration Management System (RCMS) is an extension of the Raven System designed to support structured configuration management. The principle component of RCMS is the Raven Configuration Management Language (RCML). Although RCML is distinct from the Raven Language, certain portions of Raven have been incorporated into RCML. Brief descriptions of the features of the Raven language are provided as necessary, but the reader is directed to *The Raven System* [39] for a more comprehensive presentation.

A-1.1 The Raven Language

Raven is an imperative, strongly-typed, object-oriented language with stylistic influences from C. As in most object-oriented languages, the basic unit of computing is the object. In Raven, each object is an instance of a particular class. A class encapsulates behaviour which is shared by all of its instances. The basic unit of programming in Raven is the class definition.

The Raven language is uniformly object-oriented. That is, all entities which can be manipulated in the context of Raven are objects. Objects are referred to via capabilities which may convey information on usage restrictions in addition to functioning as references.

There is only one form of computational activity in Raven: behaviour invocation. Behaviours are procedures defined within the context of a particular class. Invocation is the process of activating this procedure in the context of a particular object. Syntactically, an invocation consists of an expression identifying the object being invoked upon, the name of the behaviour being invoked, and a list of expressions which constitute the arguments provided to the behaviour. During its execution, the behaviour has access to the state of the object which was invoked upon. This state is stored in a set of instance variables which are

bound to the object. Because an executing behaviour has access to these instance variables, its execution may change the state of the object invoked upon.

The instance variables of an object are defined as part of the class definition. Although Raven supports public access to instance variables which are defined as part of the external interface of a class, such access can be regarded as syntactic sugar for an access mechanism relying on invocation of behaviours to get or set the value of an instance variable.

A-1.2 Principle Concepts of RCMS

The purpose of RCMS is to provide an environment for configuration management which is a natural extension of the object-oriented environment provided by the Raven system. The underlying tenet of RCMS is that configuration management is the process of defining and maintaining relationships between objects. These relationships may relate to many different aspects of the system: communications structure, performance, reliability, etc. In RCMS, relationships are expressed declaratively using predicate logic. Groups of related objects are bound together with configuration information to create a *composite object.* Composite objects are externally indistinguishable from those programmed in the Raven language (henceforth referred to as elementary objects). However, their internal state is governed by a collection of constraints which establish the legitimate relationships between the components of a composite object.

The basic unit of programming in RCML is the composite class definition. Like a Raven class definition, it consists an external interface declaration and an implementation. The interface of a composite class is very similar to that of an elementary class, indeed the Raven compiler and the RCML compiler both understand both styles of interface declaration. The most notable difference between a composite class interface and an elementary class interface, is that the former does not support behaviour declarations.

The implementation of a composite class, however, differs significantly from that of an elementary class. As Raven is an imperative programming language, the implementation of an elementary class consists of a series of behaviour definitions, each of which is, itself, a series of statements designed to carry out some task. The implementation of a composite class, on the other hand, consists of a specifications clause and a repair clause. The specifications clause defines constraints and the repair clause defines mechanisms to deal with situations when these constraints are violated.

The remainder of this document provides a thorough description of the syntax and semantics of the RCML language. The approach taken is *bottom-up*, that is, the language is described first in terms of its simplest elements and then in increasingly more complex structures built upon them. We begin with an overview of the notation used to describe the syntax of RCML.

A-2.0 Syntactic Notation

The grammar for RCML is expressed in an extended-BNF form. The following notations are used:

$[x]$	zero or one occurrences of x
$\{x\}$	zero or more occurrences of x
$x \mid y$	x or y
(x)	x with parentheses used for grouping.

Literal words and characters are shown in `typewriter` type. Italicized words such as *expression* represent non-terminals of the grammar. A complete production looks like:

operator $\Rightarrow$ + | −

A-2.1 Reserved Words

Reserved words are identifiers which have a unique syntactic value. Reserved words cannot be used to name variables, classes, or other elements of RCML definitions. The reserved words of RCML are:

```
all           as              behav         behaviour
cast          class           composite     constructor
controlled    copy            define        else
exists        for             if            in
indirect      link            local         lock
migrate       no              nolock        number
of            out             persistent    private
public        read            readlock      recoverable
repair        replicate       restricted    specification
strong        strongreplicate there         uncontrolled
unrestricted  volatile        write         writelock
```

Case is significant in reserved words, that is, `class` cannot be spelled `CLASS` or any other such variation. Some of these reserved words are present solely to allow the RCML compiler to process Raven header files. They are not normally used as part of RCML declarations.

A-2.2 Identifiers and Punctuation

An identifier is a sequence of letters (a-z,A-Z), digits (0-9), and underscores (_), that does not begin with either a digit or an underscore. Identifiers are used to name variables, classes, etc. RCML also makes used of a number of punctuation symbols:

```
(    )    {    }    [    ]    -->  <--> =    :=
==   ...  .    :    &    |    -    +    *    /
%    >=   <=   <    >    !=   ~    ;    ,    $
//   /*   */   "
```

Whitespace may not separate any of the characters of a punctuation symbol. When the interpretation of a sequence of characters is ambiguous, the interpretation chosen is that which results in the longest possible sequence of characters begin recognized. Thus, `<-->` is not interpreted as < followed by `-->`, but rather as the single symbol `<-->`.

The symbols `//`, `/*`, and `*/` are used for comments. Characters in-between a `//` and the end of the line on which it occurs are ignored by the compiler. Similarly characters in-between a `/*` and `*/` are ignored. The latter form of comments cannot be nested. That is, the first occurrence of `*/` terminates a comment regardless of how many un-matched `/*` precede it.

A-2.3 Simple Constants

Constants of several types may be specified in RCML. A decimal integer constant is a non-empty sequence of digits (0-9). An octal character constant (also technically an integer) is specified as a backslash (\) followed by 1, 2 or 3 octal digits (0-7). Integer constants are treated as instances of class `Int`. The identifiers `True` and `False` are defined via the preprocessor to be the integer constants 1 and 0, respectively. A string constant is a possibly empty sequence of characters enclosed in double quotes ("). Strings constants are treated as instances of class `String`. The predefined identifier `nil` evaluates to a capability for the unique "nil object." This value is used to initialize variables which hold capabilities and sometimes used as a return value to signal failure. The nil object is the sole instance of class `UndefinedObject`. The predefined identifier `system` refers to the Raven "system object." There is one system object per site and it provides access to state information about that site and to various primitive Raven operations. The system object is an instance of class `System`. The predefined identifier `self` refers to the composite object within whose definition it is found.

A-3.0 Type References and Variables

In RCML, as in Raven, a variable is a container which holds a reference to an object. A reference to an object is called a capability. The mapping of objects onto capabilities is not 1-to-1. Capabilities may also contain information restricting the manner in which the references object may be used. For example, a capability for a given object may allow the holder to carry out some operations on that object, while disallowing others. A single object, therefore, can have many capabilities.

Before proceeding to the more formal description of types and variables, it is important to note that there is a large overlap between the semantics of Raven and RCML. As was noted earlier, the design of RCML has been guided by the desire to provide seamless integration of configuration management into Raven. RCML concepts which have a parallel in Raven, thus, are expressed using the syntax and semantics of Raven. Some of the concepts presented in this manual may be dealt with quite tersely, if they represent material covered in the Raven language manual. Attempting to understand or use this manual without a Raven manual at hand, is not recommended.

RCML makes use of both typed and untyped variables. As in Raven, a type in RCML is simply the description of an interface to an object. An interface is basically a list of the operations which an object supports. Section A-9.1 provides a description of object interfaces. Assignment of a capability to a variable is constrained by the variable's type. Only capabilities referring to objects which support at least the interface specified by the variable's type may be assigned to the variable. A formal description of these *type conformance* rules is provided in Section A-7.2. Types are generally explicitly given only once and thereafter referred to by name. A type reference is:

type-ref ⇒ [*] *class-name* [[*type-ref* { , *type-ref* }]]

The simplest form of type reference is simply the name of a declared class. A class declaration is a named interface which may also have an associated implementation. There are no built-in classes in RCML, however, there is a standard library of classes which are provided with the Raven system. The file `basic.r`, in the Raven system include directory, contains the class declarations for the most elementary portions of this standard library. The file `Composite.h` provides the interface declaration for classes which are required by RCMS. The file `RCMS.h` provides the interface declaration for the RCMS system class. There is a single instance of this class per site. It provides access to various RCMS primitives. When writing an RCML class, it is necessary to include the file `Composite.h` in order to establish the appropriate programming environment. The syntax for file inclusion is the same as that used with the C preprocessor, namely:

include-stmt ⇒ `#include` (<*filename*> | "*filename*")

Conforming to standard use, the only difference between the two forms is the set of paths searched in order to find the named file.

If the class name in the type reference is preceded by an asterisk, it indicates that the referenced class is the meta-class of *class-name*. The optional bracket-enclosed list of type references following the class name are type parameters. Type parameterization is familiar to most programmers, although not in the general form in which is it available in Raven/RCML. The parameterized type common to most programming languages is the array. By itself, the term array does not denote a particular type. In order to fully specify an array, one must indicate the type of the elements which the array holds. This is type parameterization. For example, the standard Raven class library contains a class `Array`, which takes a single type parameter. The type references corresponding to an array of integers looks like:

```
Array[Int]
```

Parameterized types are described in more detail in Section A-7.1.

There are four forms of variable declaration in RCML: instance variables, local variables, behaviour parameters, and predicate parameters.

instance-var ⇒ *var-name* {, *var-name*} : *type-ref* [`public`] [`class`] [`indirect`]

local-var ⇒ `var` *var-name* {, *var-name*} : *type-ref* [= *expr*]

parameter ⇒ *var-name* {, *var-name*} : *type-ref*

predicate-parameter ⇒ *identifier*

var-name ⇒ [`copy`] [`$`] *identifier*

Predicate parameters are untyped variables and, hence, are specified simply by giving their name. Any capability may be assigned to an untyped variable and any invocation on such a variable is accepted by the compiler. Static type checking is not performed. Typing errors are detected at run-time. For some sorts of RCML expressions it is difficult to statically compute their type. Use of untyped variables makes it possible to ignore this problem for the present, at the expense of having a less satisfactory programming environment.

The general format of the other varieties of variable declaration is a list of identifiers (the names of the variables) followed by a colon and the type for the variables. Local variable definitions are always preceded by the keyword `var`. Each variable name may be proceeded by the keyword `copy` and/or the `$` symbol. These attributes are not significant within RCML. Their use is restricted to Raven class declarations. The keyword `copy` is valid only for behaviour parameter declarations. It indicates that when the behaviour is invoked, a copy of the parameter is created, and the capability for that copy is passed to the invoked behaviour. The attribute `$` indicates that only capabilities of *exactly* the type specified may be assigned to the variable/parameter. It is used as a hint to the Raven compiler to allow more efficient code to be generated.

An instance variable declaration may be followed by the keywords `public`, `class`, and `indirect`. The attribute `public` indicates that the instance variable is part of the external

interface of a class and may be directly accessed by other objects. The attribute class indicates that the instance variable should be considered to be part of the interface declaration of the meta-class of its containing class definition. This attribute is not supported for instance variables of composite classes. The attribute `indirect` indicates that the contents of the instance variable are to considered components of the composite object. In the absence of this keyword, only the object referred to by the instance variable itself is considered to be a component of the composite.

Finally, a local variable declaration may be optionally followed by the = symbol and an expression. This expression is used to assign an initial value to the variable when it first comes into scope.

RCML is a declare-before-use language. That is, the first use of a variable must occur after it has been declared. Some constructs in the language implicitly declare variables. A predicate definition implicitly declares its parameters as untyped variables. Existential quantification, universal quantification, and dynamic set constructors implicitly declare the free variable as an untyped variable.

A-4.0 Expressions

Expressions are described of the following productions:

(1) expr ⇒ `for all` *identifier* [*expr*] *expr*

(2) expr ⇒ `there exists` *identifier* [*expr*] *expr*

(3) expr ⇒ *term* {*binary-operator term*}

(4) expr ⇒ {*unary-operator*} *term*

(5) expr ⇒ *term* . *identifier* [`(`*expr* {`,`*expr*}`)`]

(6) term ⇒ *identifier* `(`*expr* {`,`*expr*}`)`

(7) term ⇒ *identifier*

(8) term ⇒ *type-ref*

(9) term ⇒ `{`[*expr*]{`,`*expr*}`}`

(10) term ⇒ `all` *identifier* `:` *expr*

(11) term ⇒ `cast` *term* `as` *type-ref*

(12) term ⇒ `(`*expr*`)`

(13) term ⇒ *integer-constant*

(14) term ⇒ *string-constant*

(15) binary-operator ⇒ `&` | `|` | `-->` | `<-->` | `<` | `>` | `<=` | `>=` | `==` | `=` | `:=` | `!=` | `in` | `+` | `-` | `*` | `/` | `%`

(16) unary-operator ⇒ `~` | `-`

A-4.1 Simple Expressions

Most of the elements of the syntax for expressions correspond to the familiar notions from mathematics. The only arithmetic type for which there is significant syntactical support in RCML is integers. RCML supports the usual variety of integer arithmetic operators with the accepted precedence rules. Primitive constants are represented by productions 13 and 14. References to variables and predefined identifiers are handled by production 7.

In addition to the normal arithmetic operators, RCML supports a number of logical operators. These are: `&` (and), `|` (or), `-->` (implication), `<-->` (logical equivalence), and `~` (negation). Each of these operators is essentially a function described by a truth table. The four binary operators take two boolean values and return a boolean value. Logical negation takes

a boolean value and returns a boolean value. A brief aside should be made at this point with regards to the notion of boolean values. In actual fact, Raven, and hence, RCML, do not support a distinct boolean type. Boolean values are represented as integers, with zero representing *false* and any non-zero value representing *true*. Nonetheless, it is convenient to speak as though a boolean type existed simply to aid in an intuitive understanding of RCML's features. McCarthy evaluation is used where applicable.

Production 5 describes both invocations and accesses to instance variables. The expression on the left hand side of the invocation operator (`.`) is evaluated to obtain the capability for the object to be invoked upon. Behaviour invocation is distinguished from instance variable access by the presence of a parenthesis-enclosed expression list following the identifier on the right hand side of the invocation operator. These are the arguments for behaviour invocation. If there are no arguments present, the identifier is the name of the instance variable, otherwise the identifier is the name of the behaviour to be invoked. For example, a class `Circle` might be used as follows:

```
aCircle.radius = 2;
area = aCircle.area();
```

The first statement sets the value of the instance variable `radius` to 2, and the second statement causes the area of the circle to be computed and assigned to variable `area`.

Production 8 is used to refer to the object which represents the named class. In Raven, classes themselves exist within the system as actual objects. The typical use of a reference to a class is the allocation of a new object. For example, to allocate a new array of integers the following expression is used:

```
Array[Int].new(0)
```

Note that this is an ordinary Raven invocation, save for the fact that the object being invoked on is a class. The parameter passed off to the invocation of `new()` is, in this case, a specification of the lower bound for the array.

Production 6 is a predicate evaluation. The identifier names the predicate to be evaluated and the parenthesis-enclosed list of expressions are its arguments.

Production 11 is a type-coercion operation. Type coercion forces the expression following the keyword `cast` to have the type associated with the class named by the identifier following the keyword `as`. Type coercion is not used frequently in RCML. Type conformance in RCML is discussed in Section A-7.2.

A-4.2 Quantified Expressions

Productions 1 and 2 represent the logical operations of universal and existential quantification, respectively. In RCML, the domain of quantification if always finite. The value of a universally quantified expression is a finite-length conjunction of terms. Each term of this conjunction corresponds to a copy of the quantified expression (at the far right hand side) with all occurrences of the quantified variable (the identifier following the keywords `for all`) replaced by the value of an element of the domain. An existentially quantified expression is a finite-length disjunction of terms constructed in a similar manner. McCarthy evaluation is used for these constructed finite-length expressions.The domain used in the evaluation of quantified expressions is the change set of the composite.

The change set of a composite is the set of all objects which could affect the state of the composite. The state of the composite, in turn, is the collection of the states of its instance variables. At first glance, it might seem therefore that the change set of a composite is nothing more than the collection of its instance variables. However, the state of an object which

is an instance variable of a composite is itself a function of the values of its instance variables, and so on. Thus, the change set of a composite consists of all objects which can be reached from the composite by traversing a chain of object references through their instance variables.

The optional bracket-enclosed expression is a *filter predicate*. The filter predicate is a boolean-valued expression. The intuitive notion is that the filter predicate is evaluated for each element of the domain. Each element for which it evaluates to true is then added to a restricted domain which is used to evaluate the quantified expression. However, if this filter predicate is *iterable*, then RCMS is able to evaluate the quantified expression considerably more efficiently.

To determine whether a filter predicate is iterable, it must be represented in disjunctive normal form. Such an expression is iterable iff each of its conjunctive terms is iterable. A conjunctive term is iterable iff at least one of its factors is iterable. Finally, a factor is iterable iff it is a non-negated use of the `in` operator in which the quantified variable appears on the left hand side. For example, suppose that `sources` is a set of source file objects, then:

```
X in sources
```

is an iterable expression and the iteration set is specifies is simply the objects obtained by iterating over `sources`. Thus, the domain of quantification for:

```
for all X [X in sources] X.tested
```

is precisely the set of source file objects contained in `sources`. If the conjunctive term contains additional non-iterable factors they are used to filter the iteration set produced by the iterable factor. So, the iteration set produced by the expression:

```
X in sources & X.tested
```

is the set of source file objects contained in `sources` which have the `tested` attribute set. The iteration set for a conjunctive term containing more than one iterable factor is the intersection of the iteration sets of each iterable factor, filtered by the remaining non-iterable factors. The iteration set for a filter expression consisting of more than one conjunctive terms is the union of the iteration sets produced by each term.

A-4.3 Set Constructors

Productions 9 and 10 are set constructors. These expressions dynamically create objects of class `IdSet[X]`. Production 9 is an enumerated set constructor. It evaluates to a set which contains all of the values in the brace-enclosed expression list. Note that the expression list may be empty, in which case the resulting set is empty. Production 10 is functional set constructor. It is similar to a quantified expression. It evaluates to a set which contains all elements of the domain of quantification for which the expression on the right hand side of the colon evaluates to *true*. As with quantified expressions the domain of quantification is the change set of the composite. If the expression is iterable, then the value of the constructed set is just the iteration set of the expression.

A-4.4 Assignment Operators

Most operators in RCML have no side-effects. The obvious exception is the invocation operator (`.`) which allows arbitrary behaviour invocation. Whether such an invocation has side-effects is determined by the semantics of the behaviour itself. RCML also supports two assignment operators. Conceptually, these two assignment operators can be conceived of as an equality constraint (==) associated with a specific repair mechanism. Temporal assignment (:=) causes the value on its right hand side to be assigned to the instance variable specified on its left hand side each time the composite is validated. Initial assignment (=) causes the values on its right hand side to be assigned to the instance variable specified by its left

hand side just once, when the composite is first allocated. Initial assignment is not permitted in repair scripts, while the = operator simply indicates a normal assignment statement as in Raven.

A-5.0 Predicate Definitions

Predicates may only be defined in the context of the specification clause of a composite declaration. The syntax for a predicate definition is:

predicate-definition ⇒ `define` *ident* `(`[*ident* {`,`*ident*}]`)` `<-` *expr*

The identifier following the keyword `define` is the name of the predicate. The parenthesis-enclosed list following the predicate name is its argument list. Each identifier of the argument list is declared as an untyped variable whose scope is the expression on the right hand side of the <- symbol. The expression itself, which is called the *body* of the predicate, must be boolean valued. When a predicate is evaluated, the supplied argument values are bound to the argument variables, and the body of the predicate is evaluated.

A-6.0 Statements

In the context of repair scripts a subset of the Raven language is used. This section provides a brief description of the those features of Raven. The syntax of a statement is:

(1) statement ⇒ *expr*

(2) statement ⇒ `if (`*expr*`)` *block* [`else` *block*]

(3) statement ⇒ `while (`*expr*`)` *block*

(4) statement ⇒ `for (`[*expr*]`;`[*expr*]`;`[*expr*]`)` *block*

(5) block ⇒ *statement*

(6) block ⇒ `{`[*statement*]{`,` *statement*}`}`

These statements are described completely in the Raven language manual. Their meaning should be familiar to anyone acquainted with imperative programming languages. It should

be noted, however, that the expressions used in these statements are those described in Section A-4.0, not the expressions defined for Raven. Thus, in RCML,

```
aSet = all X : X in sources & X.tested
```

is a valid statement, which when executed, evaluates the functional set constructor specified on its right, and assigns it to the variable `aSet`.

A-7.0 Class Declarations

RCML supports the full syntax of Raven class declarations in order to permit the type safe use of elementary objects in RCML composites. The syntax for class interface declarations is:

class-decl ⇒ [`public`] `class` *ident* [[*ident* { , *ident*}]] [<- *type-ref*] [*properties*]
{{*instance-var*; | *behaviour-decl*; | *in-link-decl*; | *out-link-decl*; }}

Only those elements of the interface declaration which are directly relevant to RCML are described here. A complete description of class interface declarations is found in the Raven language manual. The purpose of the class interface declaration is to describe the ways in which an instance of that class can be manipulated. A simple class declaration for a stack of integers looks like:

```
class Stack
    {
    size : Int public;
    behav push(item:Int);
    behav pop() : Int;
    }
```

This interface indicates that a stack object has a publicly accessible instance variable called size and two behaviours called push and pop. Although the interface does not specify semantics, good programming style would dictate that these behaviours modify the stack object in the manner suggested by their names.

The body of the class interface declaration consists of four types of declarations: instance variables, behaviours, inbound links, and outbound links. The syntax of instance variable declarations is given in Section A-3.0. Instance variable attribute `indirect` cannot be used in a class interface declaration. The syntax of the remaining three interface elements are given by:

(1) behaviour-decl ⇒ [`copy`] `behav` *ident* (*[ident:type-ref]{, ident:type-ref}*) [*:type-ref*]
[`read lock` | `write lock` | `no lock`][`private`][`class`]

(2) behaviour-decl ⇒ *constructor-decl*

(3) constructor-decl ⇒ `constructor` (*[ident:type-ref]{, ident:type-ref}*) [*class*]

(4) in-link-decl ⇒ `link in` *ident* (*[ident:type-ref]{, ident:type-ref}*)

(5) out-link-decl ⇒ `link out` *ident* (*[type-ref]{, type-ref}*)

A behaviour is a procedure which executes in the context of an object. As with a procedure in a typical imperative programming language, it has a name (following the keyword `behav`), a list of parameters and the type of those parameters (enclosed in parentheses) and an optional return type. The second form of behaviour declaration (beginning with the keyword `constructor`) defines an initialization behaviour which cannot be directly invoked from within Raven. Rather it is invoked by the Raven system when a new object of the given class is allocated. The constructor is responsible for initializing the state of the newly allocated object. Behaviour declarations support a number of attributes which are concerned with concurrency control, visibility etc.

Productions 4 and 5 are inter-object link declarations. Links are an alternative invocation mechanism in which the invoker does not explicitly identify either the target object or the behaviour being invoked. Instead, these values are bound to the link dynamically via a special system link operation. Data is sent on an *outbound* link and is received by an *inbound* link. Data sent via a link is typed. An inbound link declaration contains a parameter declaration list much like that of a behaviour declaration. An outbound link declaration has a

slightly different parameter list, in which only the parameter types are listed. For a connection to be established between an inbound link and an outbound link, they must both have the same number of parameters and the type of each outbound parameter must be compatible with the type of the corresponding inbound parameter.

In Raven, the `send` statement is used to transmit data on a link. The values supplied with the send statement are the name of the outbound link via which the data is to be sent and a set of values corresponding to the parameters declared for that link. Data arriving at an inbound link is passed on to a link handler routine, which is identical to a behaviour save for the fact that a return value is not permitted.

Links are used to create *configuration-independent* objects. Configuration independence means that the environment in which an object can operate is not statically determined by its implementation. Using links, communication between objects always occurs via locally named end-points, that is, the inbound and outbound links. There are no references to external objects and, hence, it is not possible to embed static configuration information in an object. The resulting system is similar to the familiar pipes and filters paradigm of Unix, but is more flexible because objects may have many links, rather than just a standard input and a standard output as in Unix.

A-7.1 Type Parameterization

In the previous section, the class declaration for a simple stack of integers is given. A stack is a fairly generic data structure, and there is much in common between a stack of integers and a stack of strings. Raven uses the notion of type parameterization to capture this com-

monality. Consider the class declaration for a more generic stack which uses type parameterization:

```
class Stack[X]
    {
    size : Int public;
    behav push(item:X);
    behav pop() : X;
    }
```

In effect, this declares a collection of class interfaces. For example, by binding the type reference `Int` to the parameter `X`, the class declaration of the previous section is obtained. Type binding occurs by replacing the type parameter with a concrete type reference. The type reference for a stack of integers would be:

```
Stack[Int]
```

and,

```
Stack[List[String]]
```

would be the type reference for a stack of lists of strings.

A-7.2 Type Compatibility

Type checking in Raven and RCML is based on the notion of compatibility. An object is compatible with its intended use if its interface supports the set of features required in that context. Objects are always manipulated via variables, and compatibility is, thus, expressed in term of types. A type `T1` is compatible with type `T2` iff:

(1) For each instance variable `V2` of `T2`, there exists an identically named instance variable `V1` of `T1`. The types of `V1` and `V2` must be mutually compatible. The attributes of `V1` and `V2` must be identical.

(2) For each behaviour `B2` of `T2`, there exists an identically named behaviour `B1` of `T1`. The number of parameters in `B1` and `B2` must be identical. The names of the parameters must be identical by position. The type of each parameter of `B2` must be compatible with the type of the corresponding parameter of `B1`. The type of the return value of `B1` must

be compatible with the type of the return value of `B2`. The attributes of `B1` and `B2` must be identical. The exception to this rule is the `constructor`, which may differ both in the number and in the types of its parameters.

(3) For each inbound link `L2` of `T2`, there exists an identically named inbound link `L1` of `T1`. The number of parameters in `L1` and `L2` must be identical. The names of the parameters must be identical by position. The type of each parameter of `L2` must be compatible with the type of the corresponding parameter of `L1`.

(4) For each outbound link `L2` of `T2`, there exists an identically named outbound link `L1` of `T1`. The number of parameters in `L1` and `L2` must be identical. The type of each parameter of `L1` must be compatible with the corresponding parameter of `L2`.

Type comparison for parameterized types is performed by carrying out the specified type bindings to create a *virtual* interface declaration, and then applying the above rules.

Raven also supports a special type called `cap`, which stands for "capability" and effectively represents a reference to an object of arbitrary type. The type `cap` is compatible with all other types, and all other types are compatible with it. Invocations on variables of type `cap` cause a run-time type checking mechanism to be used.

A-8.0 Standard Features of Raven

In an abstract sense, there are no built-in features in Raven. All behaviour is encapsulated in the class library. However, Raven is supplied with a standard class library which establishes the "normal" Raven environment. This class library includes primitive data types such as integers, characters, and strings, as well as useful abstract types such as lists, stacks, and arrays. The elements of the class library most directly relevant to RCML are the link manipulation behaviours. These behaviours belong to the `System` class. The Raven run-time ensures that each Raven site possesses exactly one instance of the `System` class, which is

naturally enough referred to as the *system object*. The link manipulation routines supported by the system object are:

```
makeLink(src:Object, sLName:String, dest:Object, dLName:String)
unLink(src:Object, sLName:String, dest:Object, dLName:String)
unLinkAll(src:Object, slName:String)
linkExists(src:Object, sLName:String, dest:Object, dLName:String)
linkLive(src:Object, sLName:String, dest:Object, dLName:String)
numOutputs(src:Object, sLName:String)
numInputs(dest:Object, dlName:String)
```

Behaviour `makeLink()` establishes a connection between the outbound link `sLName` of object `src` and the inbound link `dLName` of object `dest`. Behaviour `unLink()` terminates such a connection. Behaviour `unLinkAll()` terminates all connections associated with outbound link `slName` of object `src`. Behaviour `linkLive()` determines whether it is still possible to send data across the indicated connection, that is, it checks for communication failure. Behaviour `numOutputs()` returns the number of connections emanating from outbound link `sLName` of object `src`. Behaviour `numInputs()` returns the number of connections arriving at inbound link `dLName` of `dest`.

A-9.0 Composite Class Definitions

Programming in RCML consists of creating *composite class* definitions. There are a number of parallels between a composite object and an elementary object. In particular, with respect to the way in which it is manipulated, a composite object is indistinguishable from an elementary object. As such, the interface declaration for a composite is nearly identical to that given in Section A-7.0. In fact, the Raven compiler is capable of accepting composite class interface declaration in order that composite object may be allocated and used by elementary objects. The implementation of a composite object differs substantially from that of a elementary object. Whereas, the implementation of an elementary class consists of behaviour and link definitions written in an imperative language, the implementation of a composite object consists of a *specification clause*, which defines constraints over the elements of the

composite, and a *repair clause*, which defines how to address violations of these constraints. The syntax for a composite class definition is:

composite-def ⇒ *composite-decl specification-clause repair-clause*

A-9.1 Interface Declarations

The syntax of a composite declaration is:

composite-decl ⇒ `composite` *ident*
{{*instance-var*; | *in-link-decl*; | *out-link-decl*; | *constructor-decl*;}}

It is quite evident that the form of a composite declaration is considerably simpler than that of an elementary class. The current version of RCML does not support inheritance or type parameterization for composites. The attributes of a composite class are currently determined automatically by the compiler based on the nature of the composite.

Note also, that behaviour declarations are not allowed. The motivation for this is the lack of any means to implement behaviours given the non-procedural nature of the implementation of a composite. This reasoning should also, however, disallow the declaration of links and the constructor. These two items are used somewhat differently in RCML than in Raven.

Allowing the constructor declaration provides a mechanism for parameterizing composite objects at the time of their creation. The parameters declared in the constructor of a composite are made available as instance variables of the composite, so that they may be referred to in the context of the specification or repair clause.

Allowing link declarations provides a mechanism for creating configuration-independent composite objects. Links in a composite object act as gateways between the system outside of a composite and the components which constitute its internal structure. Each inbound link of a composite object has a corresponding outbound link of the same name which is visible only to the components of the composite. Data arriving on the inbound link is automatically

forwarded to this outbound link. Thus, if this outbound link is connected to the inbound link of an object inside of the composite, all data sent to the composite is forwarded along to that object. In a similar manner, and outbound link of a composite has a corresponding inbound link which is visible only to components of the composite. Data sent to this outbound link is automatically forwarded to the outbound link of the composite.

Forwarding links such as these allow the isolation of the internal structure of the composite from its external interface. The internal communication structure of the composite is invisible, and indeed, may change over time as a result of re-configuration, without in any way affecting the external interface of the composite.

A-9.2 Component Set

One of the key elements of the definition of a composite object is that at any given time, it consists of a well-defined set of objects which are referred to as the components of the composite object. The set of components of a composite object is called its *components set.*

The components set of a composite object is derived from its instance variable declarations. The keyword `indirect`, which is allowed only on instance variable declarations in a composite, defines the nature of this derivation. If the keyword `indirect` is absent on an instance variable declaration, then the object referred to by that instance variable is defined to be a member of the components set. If the keyword is present, then the components set is defined to contain the objects obtained by iterating over the object referred to by the instance variable.

Use of the `indirect` attribute requires that the instance variable support iteration. This means that the type of the instance variable must support the behaviour `getIterator()` with a return type of `Iterator[X]`, where X is the type of the objects generated during

iteration. For example, if the stack of integers class of Section A-7.0 has its interface extended to:

```
class Stack
    {
    size : Int public;
    behav push(item:Int);
    behav pop() : Int;
    behav getIterator() : Iterator[Int];
    }
```

then the type `Stack` would qualify as iterable. The standard Raven library provides an assortment of classes, all of which inherit from class `Collection[X]`, which support iteration. The classes include `List[X]`, `Array[X]`, `Set[X]`, and `IdSet[X]`.

The latter of these is quite useful in composite object definitions. It offers all the usual features of sets, but set membership is based on object identity rather than object value, as is the case with class `Set[X]`. The difference between the two is most easily demonstrated by a simple example. Consider the following code fragment:

```
var aSet:Set[String] = Set[String].new();
var anIdSet:IdSet[String] = IdSet[String].new();
var aString1:String = "hello";
var aString2:String = "hello";
aSet.add(aString1);
aSet.add(aString2);
anIdSet.add(aString1);
anIdSet.add(aString2);
```

After the execution of this code, `aSet` contains only one item, while an `anIdSet` contains two. When attempting to add `aString2` to `aSet`, its value is compared to the existing member of the set using the behaviour `equal()`. The precise semantics of the behaviour `equal()` are defined by the class implementer, but intuitively it is intended to compute whether two objects represent the same value. As both strings consist of the same sequence of characters, this comparison returns *true*, and `aString2` is not added to the set. When attempting to add `aString2` to `anIdSet`, the comparison is based on object identity.

Although `aString1` and `aString2` both represent the same string, they are nonetheless, distinct objects, and the result is that `aString2` is added to `anIdSet`.

The components set of a composite need not be fixed over time. The use of the temporal assignment operator in the specifications clause may cause the value of a composite's instance variables to change. Furthermore, the objects referred to by instance variables with the `indirect` attribute may have their contents altered, resulting in a change in the components set. The components set for a composite is computed prior to validation of the composite. Section A-10.0 describes the events which lead to validation.

A-9.3 Specification Clause

The syntax of the specification clause of a composite class definition is:

(1) specification-clause ⇒ ({*predicate-definition*; | *constraint-definition*;})

(2) constraint-definition ⇒ [*integer-const*:] *expr*

The specification clause consists of a sequence of predicate and constraints definitions. A constraint definition is simply a boolean expression, which is optionally preceded by an integer label. This label is used to co-ordinate validation of the composite with repair of the composite when repair scripts are used. This is described in Section A-9.4.

Validation of a composite object consists of evaluating the constraints given in its specification clause. If any of the constraints evaluates to *false*, then the composite is said to be invalid with respect to its specification. When a composite is found to be invalid, RCMS invokes a repair mechanism, as described in Section A-9.4, to restore the composite to a valid state. The events which cause validation of a composite to occur are described in Section A-10.0.

Prior to the start of validation, RCMS obtains exclusive locks on all objects in the components set of the composite. This ensures that the constraints of the specification clause are evaluated in a consistent context. Constraints using the permanent or initial assignment operators are evaluated prior to other constraints, since they may affect the components set of the composite. Locks obtained prior to validation are held until after the repair process is complete.

A-9.4 Repair Clause

The syntax of the repair clause of a composite class definition is:

repair-clause ⇒ `repair {recoverable;` | {*integer-const*: *block*}}

There are two distinct forms of the repair clause, corresponding to the two forms of recovery supported by RCMS. *Roll-back recovery* is indicated by a repair clause which contains the keyword `recoverable`. In order to use roll-back recovery, all of the components of the composite must have the recoverable property. This guarantees that all modifications to any component of the composite occur within the context of a *recovery block.* A recovery block is similar to an atomic transaction in that all changes made to recoverable objects in the context of a recovery block may be aborted. However, recovery blocks do not provide permanence or concurrency control. These issues are addressed orthogonally in the Raven system. The mechanism used for roll-back recovery is simple: if the actions of a recovery block cause a composite object to become invalid, then that recovery block is aborted. Assuming that all components of the composite object are recoverable and that the composite started in a valid state, aborting the recovery block returns it to that valid state.

Roll-forward recovery is indicated by a recovery block which contains a series of repair scripts. Each repair script consists of an integer label followed by a sequence of statements in a subset of the Raven language (described in Section A-6.0). When validation of a com-

posite fails, one of the constraints which evaluated to *false* is selected, and the repair script with the corresponding label is executed. If there is no corresponding repair script a run-time error is generated. Following execution of the repair script, the locks on the components of the composite are not immediately released. Instead, the composite is re-validated and if necessary further repair scripts may be executed. It is possible that the process of validation and repair is non-terminating (indicating an error in the design of the repair scripts). RCMS terminates the validation/repair sequence after a fixed number of iterations. The number of iterations is an implementation-dependent value. A run-time error is also generated by this event.

A-9.5 Standard Features of RCML

As with Raven there are few built-in features of RCML. Currently, the only ones provided are simple syntactic sugar. The built-in predicates:

```
Connected(), Outputs(), Inputs()
```

correspond directly to the behaviours `linkExists()`, `numOutputs()`, and `numInputs()` of the system object.

A-10.0 Composite Object Monitoring

RCMS offers several different alternatives for determining what events cause a composite object to be validated. For composites which use roll-back repair, monitoring is integrated with the repair mechanism itself. Validation occurs when a *top-level recovery block* is exited. A top-level recovery block is similar to a top-level transaction, but it is initiated automatically by the Raven run-time system in response to the invocations which a thread of control makes.

As a thread of control executes within Raven it *enters* and *exits* objects. When a behaviour is invoked on a particular object, that object is entered. When that behaviour returns control to the point of invocation, the object is exited. Nested invocations result in sequences of objects being entered, then exited. If one looks at the set of objects which have been entered by a particular thread, either none of the objects is recoverable, or there is a first object which is recoverable. Upon entering this first recoverable object, the thread is said to have entered a top-level recovery block. Exiting from the object corresponds to exiting the top-level recovery block.

Composites using roll-forward repair may use one or more of three varieties of monitoring: periodic polling, direct object monitoring, and communication failure notification. The type of monitoring used for a given composite is determined at run-time by invoking certain behaviours of the RCMS object:

```
behav pollComposite(aComposite:Composite, interval:Int)
behav monitorChange(aComposite:Composite)
behav monitorFailure(aComposite:Composite)
```

Invocation of behaviour `pollComposite()` causes RCMS to validate `aComposite` every `interval` clock ticks. The actual duration of a clock tick is a system dependent value.

Invocation of behaviour `monitorChange()` causes RCMS to tag all components of the composite for direct monitoring. The invocation sequence for an object tagged for direct monitoring is altered so that RCMS is able to make a snap-shot of the object's instance variables prior to execution of the invoked behaviour. Following the execution of the invoked behaviour, RCMS compares the values of the instance variables with the snap shot. If the values differ, then any composites to which that object belongs are validated. If the compo-

nents set of the composite changes over time, the set of objects tagged for direct monitoring is updated.

Invocation of the behaviour `monitorFailure()` causes RCMS to watch for communications failures involving inter-object links of any of the components of `aComposite`. When a communications failure occurs, the communications sub-system signals RCMS. RCMS determines if the failure is connected with the sending of data on an outbound link belonging to an object which is a component of a composite. If so, the composites to which that object belongs are validated.

A-11.0 RCML Compiler

The current version of the RCML compiler is a program named `rcfg` which resides in the directory `/dsrg/raven/bin`. It is currently compiled only for the Sparc architecture. The compiler is invoked with a command line of the form:

```
rcfg [options] [files]

   where the options are:
      -v print version number of compiler and exit
      -p pre-process files and exit
      -r compile to Raven only
      -c compile to C only
      -k do not delete intermediate files
      -I use the given path to search for include files
      -L use the given path to search for libraries
      -l include the given library during the linking phase
      -g generate gdb debugging information (implies -k)
      -o use the given name for the name of the output file
      -t be verbose while compiling

   where the files are:
      extension .cfg are RCML source files
      extension .r are Raven source files
      extension .c are C source files
      extension .o are object files
```

Raven files used with the `rcfg` command are passed along to the Raven compiler when it is executed by `rcfg`. C files used with `rcfg` are passed along to the C compiler when it is

executed by `rcfg`. Object files used with `rcfg` are passed along to the C compiler when it is executed by `rcfg`. The default output file name is `rv.out`.

The program `rcfg` is actually just a driver. It parses the command line and then executes a number of other programs to actually carry out the work. RCML programs are translated to Raven by `RavenCfgTrans`. Raven programs are translated to C by `rvc`. C files are compiled by the GNU C compiler `gcc`, which is also responsible for linking all the resulting object files in order to produce an executable. All of these programs may be found in `/dsrg/raven/bin`.

Compilation of RCML programs by themselves cannot produce an executable. There is no concept of a main program or initialization sequence in RCML. To actually use a composite object defined in RCML, one needs a Raven program which allocates and makes use of that composite. One of the easiest ways to experiment with RCML is to link the object files for composite objects with the Raven shell library. The executable produced in this case will begin an interactive Raven shell session when run. From within this session, the user may allocate and generally have a good time with composite objects written in RCML.