Distributed configuration management using composite objects and constraints

To cite this article: T Coatta and G Neufeld 1994 Distrib. Syst. Engng. 1 294

View the article online for updates and enhancements.

Related content
- Implementing configuration management policies for distributed applications
  Gerald Krause and Martin Zimmermann
- Synchronizing shared objects
  M Riveill
- DC++: distributed object-oriented system support on top of OSF DCE
  A B Schill and M U Mock

Recent citations
- H P Raghunandan and S Bedekar
- T Preuss et al
- H Fossa and M Sloman
Distributed configuration management using composite objects and constraints

Terry Coatta† and Gerald Neufeld‡
Department of Computer Science, University of British Columbia, Vancouver, BC, Canada V6T 1Z2

Abstract. Distributed programming has transformed applications into federations of cooperating components. Complex interactions between such components create complex interdependencies which are outstripping the capacity of systems managers to monitor and control them. Furthermore, adding configuration management features to an application's components reduces flexibility and portability. The Raven Configuration Management System (RCMS) addresses these problems by handling configuration management orthogonally to the task of programming components. RCMS groups related components together, and provides rules which govern their interactions. RCMS combines object-oriented structuring with declarative programming to improve reliability and performance in the presence of an evolving environment. An initial version of RCMS has been completed, including a compiler and run-time support routines.

1. Introduction

Configuration management occurs in many guises in the context of modern computer systems: network management, systems administration, software development environments, and software component composition all rely heavily on configuration management services. The most notable aspect of current configuration management tools is that they are generally targeted at a very specific domain. For example, the Unix utility make [5] assists in program building and rebuilding, while the Conic system [10] specializes in dynamic re-arrangements in the communications patterns of software components. Typically, each tool uses a different syntax and semantics for expressing configuration information. RCMS approaches configuration management in a very general way, using a single formalism for expressing many different types of configuration management information.

In the broadest sense, configuration management is the control of interactions between the components of a system. This comprises three distinct tasks: defining acceptable configurations, monitoring configurations for departures from the defined limits, and correcting configurations for which such a departure has occurred. A distributed mail service, for example, may consist of components which handle interactions with users, others which handle the storage of mail, and still others which handle its exchange between separate computer systems. Each component performs certain tasks and responds to external events which modify its behaviour. Mail routers forward mail, moving it closer to its destination. Altering routing tables in a mail router may improve performance or may cause loss of connectivity in the mail system. Avoiding loss of connectivity requires that routing tables be consistent. Achieving improved performance requires monitoring and adapting to changes in the environment. A configuration management system provides a framework in which these tasks can be carried out.

Initial work on configuration management addressed these issues by providing a framework for storing and retrieving configuration information, recording the structure of applications and allowing the effects of configuration changes to be explored [8, 9, 15, 16]. Languages and systems such as Conic [10, 12] were developed to make reconfiguring software components as easy as possible. With these basic elements in place, tools for describing configuration changes and for executing those changes on running systems [6, 11] were developed. This has led naturally to a focus on automating the process of detecting the need for reconfiguration and carrying it out. The Raven Configuration Management System (RCMS), the Network Management System [17], GRAPPLE [7] and Darwin [13] all arise from an exploration of these issues.

RCMS provides a framework for configuration management which addresses three major shortcomings in existing configuration management software. First, RCMS is a general-purpose configuration management tool. It provides a single language for expressing a wide variety of configuration management information.
For example, the facilities of a tool such as make are provided using the same underlying concepts as those used for providing network management services. Second, RCMS provides automated configuration management. Rather than relying on human intervention to problems which arise during the operation of the system, RCMS monitors the system in real time, detects situations in which the system is not acting in a correct manner, and applies remedial actions to eliminate the problem. Thus, many tedious management tasks can be handled completely by the system itself. Third, RCMS was designed specifically for operation in a distributed system because the need for automated configuration management is particularly pressing in that environment.

One of the unique features of RCMS is a construct called the composite object. A composite object provides a way to group related objects together along with configuration management information which governs the interactions of those objects. In effect, the composite object is a special purpose configuration management agent whose task it is to monitor and repair a given set of objects. Each composite object implements a set of operations specifically tailored to the tasks required for managing the objects which are part of the composite, allowing them to be reasonably lightweight and effective. Another benefit of composite objects is that they are externally indistinguishable from the conventionally programmed objects of the system and, consequently, provide a seamless integration of configuration management into the underlying system.

The remainder of this paper consists of four sections. In section 2, the Raven configuration management system is described and a sample use of the system is presented. In section 3, the implementation of RCMS is discussed, emphasizing the techniques required to produce effective performance despite the generality of the RCMS configuration language. Section 4 briefly compares RCMS with several existing configuration management systems and section 5 consists of concluding remarks.

2. The Raven Configuration Management System

2.1. The Raven System

The Raven Configuration Management System is a part of the Raven Project [1, 4] at the University of British Columbia. The Raven Project is exploring a variety of issues in distributed object-oriented systems. RCMS adds configuration management capabilities to the Raven system. While certain features of Raven (e.g. its uniform object model and transparent distribution of objects) eased the implementation of RCMS, none of its features are crucial to the underlying techniques developed. The core of RCMS is a compiler which translates composite object definitions into an intermediate language (currently Raven). Retargeting the RCMS compiler would be a relatively straightforward task. Monitoring and modifying the components of the system are currently achieved using the Raven invocation mechanism which could easily be handled via other existing configuration protocols such as SNMP [3].

Because RCMS currently exists in the context of the Raven system, a short introduction to Raven will ease the task of understanding RCMS. Although the Raven language is uniformly object-oriented, it follows the style of the C language in many respects [1]. The basic unit of programming within Raven is the class. A class definition consists of a name identifying the class and a list of instance variables and behaviour declarations which define the interface of that class. This interface definition is optionally followed by a list of behaviour definitions which implement behaviours declared for the class. Here is how the interface for the prototypical stack of integers class appears in Raven:

```java
class Stack {
    data: Array[Int];
    behav Push(item: Int);
    behav Pop(): Int;
}
```

The first line of the interface defines an instance variable data used to implement the stack. Instance variables hold references to other objects. These references are called capabilities. The keyword behav names an operation and provides its signature (the return type and the names and types of its parameters).

Raven supports an additional feature useful for configuration management which is not found in other object oriented languages—the inter-object link. A link is a mechanism for exchanging data between two objects without requiring either object to be internally bound to the other. Links are an extension and refinement of the 'filters and pipes' paradigm used to connect independent processes together in Unix. The use of inter-object links in the construction of reconﬁgurable software has been extensively explored within the Conic system [14]. The key property of a link is that within an object a link is referred to only via a local name. Since objects are not referred to directly, links allow external reconfiguration of objects at run-time.

2.2. Composite objects

In RCMS, configuration management is carried out by defining and creating composite objects. Each composite object binds together a particular set of managed objects, the configuration information associated with those managed objects, and a set of configuration management services tailored to the specific needs of the managed objects.

Clearly, a central part of the definition of a composite object is the specification of configuration information. In existing systems, such configuration information is frequently expressed in terms of constraints on certain attributes of the objects or on relationships between objects. It may be undesirable, for example, for a router
in a mail system to be connected to another router which has exhibited a relatively high degree of congestion. Assuming that a mail router possesses an attribute which indicates the level of congestion, expressing this condition in a declarative manner is straightforward. The Unix utility make also relies upon constraints. For example, if file A is produced from file B, then the date of last modification of file A is constrained to be greater than the date of last modification of file B.

Examples such as this combined with the results of other researchers investigating similar problems [7,13, 17], point towards the use of a declarative language for specifying configuration information within a composite object. It should be noted that the purpose of this language is not the specification of how to provide the service that the composite object embodies—the components of the composite collectively provide that service. Rather, the specification addresses management issues such as the communications structure of the components, maintenance of performance at desired levels, and the detection of and response to failures of particular components.

The declarative nature of the configuration information which must be associated with a composite object stands in contrast to the procedural nature of typical object-oriented programming languages. Nonetheless, in order to take advantage of the familiarity of the object-oriented paradigm and to ease integration between programming in Raven and carrying out configuration management in RCMS, the definition of a composite object parallels the definition of an elementary object in Raven. The basic unit of definition in RCMS is, thus, the composite class. A composite object is simply an instance of a composite class.

The definition of a composite class consists of three parts†: first, an interface clause which defines the external features of the composite, second, a specification clause which defines the relationships and constraints between the object which are part of the composite, and third, a repair clause which defines the management operations associated with the composite.

Like an elementary class definition in Raven, a composite class definition begins with an interface clause. This interface clause defines the externally observable features of the composite. These externally visible features consist of inter-object links and instance variables. Inter-object links are the means by which other objects communicate with the composite. The instance variables hold capabilities for objects which are part of the composite, also referred to as components of the composite. In addition, the instance variables defined for the composite are used to identify the set of objects which are to be managed by the composite.

The interface clause is followed by a specification clause. The specification clause consists of two types of statements: predicate definitions and constraint expressions. Both are defined using first order predicate logic, including universal and existential quantification.

† A formal syntax is not given in this paper. Appendix A of [4] provides a detailed description of the language.

class SymbolTable {
  behav add(key: String key, item: Object);
  behav lookup(key: String): Object;
  behav containsItem(item: Object): Int;
} 

composite DirTree {
  constructor(SymbolTable : Root);
  root : SymbolTable;
  nodes : Set[SymbolTable] indirect;
} 

specification {
  define Path(X, Y) <= Y in X |
  there exists Z [Z in X] |
  Z.instanceof() == SymbolTable & Path(Z, Y);
  1: root = Root;
  2: nodes := all X : Path(root, X);
  3: for all X -Path(X, X);
} 

repair {
  atomic;
}

Figure 1. Composite object declaration for a directory tree.

The state of the composite object may also be accessed within predicates or constraints by referring to the instance variables of the composite. A composite object is said to be valid if all of the constraints in its specification clause evaluate to true.

The final element of the composite class definition is the repair clause. The purpose of the repair clause is to specify how to transform a composite object from an invalid state to a valid one. RCMS supports two forms of repair: atomic repair, which is described in section 2.3, and script-based repair, which is described in section 2.4.

2.3. A simple example

Figure 1 contains the definition of a simple composite object, as well as the interface declaration, for the Raven objects of which it is composed. This pedagogical example provides a context in which to illustrate most of the features of RCMS; section 2.4 (figure 2) describes a more realistic, but less extensive, application of RCMS.

The composite object of figure 1 is a directory tree composed of a collection of symbol table objects. Each symbol table is a set of key/value pairs in which the values are capabilities referring to arbitrary objects. Values which refer to other symbol tables create a directed graph of symbol tables. Following a given path through the graph generates a sequence of keys which can be regarded as a path name much like the ones used in Unix file systems. The configuration management activities associated with this directory tree consist solely of ensuring that no modification of the tree results in cyclic references within the tree.

The interface of the composite class definition begins with the keyword composite. Composite interfaces contain a list of instance variables and a list of inter-object link definitions. In addition, the interface may contain a constructor() definition which permits
The composite to a valid state. The directory tree is a predicate defining connectedness between symbol tables. A path exists between two symbol tables if one is an entry of the other or if there is a sequence of such symbol tables, each one referring to the next element of the sequence. A cycle exists if any table within the tree has a path to itself. The operator use the interface declaration contains the most interesting items. The keyword indirect on the declaration of the instance variable nodes. This keyword indicates that the contents of nodes are to be considered as components of the composite object, rather than the set nodes itself. Thus, the components of this composite are the root of the directory tree, stored in root, and all of the remaining nodes of the directory tree, stored as elements of nodes.

For this example, the specification clause which follows the interface declaration contains the most interesting items. The keyword indirect on the declaration of the instance variable nodes. This keyword indicates that the contents of nodes are to be considered as components of the composite object, rather than the set nodes itself. Thus, the components of this composite are the root of the directory tree, stored in root, and all of the remaining nodes of the directory tree, stored as elements of nodes.

The constructor for this composite takes a single parameter. This parameter is the root of the directory tree and needs to be stored in the instance variable root. Constraint 1 of the specification achieves this. The := operator means that root should initially be equal to Root, the parameter to the constructor. The default mechanism for satisfying this constraint in RCMS is simply to assign the value to the appropriate instance variable when the composite is created.

Constraint 2 uses the := operator, which means that the value of nodes is always constrained to have the value specified by the set constructor to its right. The specified set consists of all symbol tables reachable from the root. This set is, thus, precisely the set of symbol tables which constitute the directory tree minus the root. The := operator ensures that the value of nodes will be updated to reflect any changes to the directory tree.

Constraint 3 is the core of the composite definition. It states that there are no cycles in the directory tree. Through real-time monitoring of the composite object's state, RCMS detects when any of these constraints have been violated and invokes repair mechanisms to restore the composite to a valid state. The keyword repair introduces the third part of the composite object definition, which specifies the mechanism used to restore the composite to a valid state. The directory tree uses atomic repair. Components of an atomic composite must always be accessed within the scope of an atomic transaction. Such a composite object can be returned to a valid state simply by aborting the transaction which created the invalid state. This example has been implemented and tested using the current RCMS implementation.

2.4. A mail transport system
While atomic repair is useful when modifications to a system can simply be aborted, configuration management often demands adapting to changes in the environment rather than simply disallowing them. Figure 2 is a composite object definition for a mail handling system (MHS) which illustrates using RCMS in this manner.

The purpose of the MHS composite is to receive mail items, sort them according to their priority, and then direct them to a transport mechanism for delivery to another computer system. The Raven definitions for the component objects used in this composite are given in figure 3.

The interface for this composite object is much simpler than the one for the directory tree. No use is made of parametrization via the constructor method. The set of components of the composite can be statically computed at compile time because that is the only time that the instance variables of the composite are altered. Inter-object links connect the components internally and provide the external interface of the MHS composite.

The MHS composite has a single input link on which mail items arrive. This connects to the input link of the object sorter which sorts mail items according to their priority and sends them over one of two output links, highPrio and lowPrio. These output links connect to the input link of an appropriate mail transport object, or to the input link of the buffer object hold. The communication structure of the composite object is illustrated in figure 4. In this diagram, all possible output paths from the sorter object to the mail transport objects and buffer object are shown. During the operation of the system, however, each output of the sorter object would be connected to exactly one other object.

The basic configuration management activity of the MHS composite is to ensure that mail items are delivered to a suitable transport service whenever possible. Two factors govern the suitability of a given mail transport service. First, each mail transport service charges a certain amount for the delivery of a mail item. This amount is indicated by the cost attribute of the transport service object. A low priority mail will not be sent via a particular transport service if its cost is too high. Second, other aspects of the configuration are static as well. For example, some of the inter-object links do not evolve over the lifetime of the composite. Isolating static elements of a composite object definition could offer dramatic performance advantages for RCMS, but it is an issue which has not been investigated thus far.
The mail transport service itself may be unavailable (for example, the phone system may be down). Each of the transport mechanisms has an attribute `up` which indicates whether it is able to deliver mail. If no suitable transport mechanisms are available for a given mail item, then it

must be directed to the buffer object which holds it for future delivery.

The specification clause of the MHS composite definition begins with predicate definitions which describe when a mail transport object may be used, and whether there are any suitable mail transport objects available, for mail items of high and low priority respectively. These predicates are followed by a series of constraints governing the interconnections of the various components.

As indicated in the composite declaration, the external interface of the MHS composite consists of a single input link which accepts mail items. These

- **Figure 2. A mail handling system.**
- **Figure 3. Component object definitions for the mail transport composite.**
- **Figure 4. Structure of mail handling system.**
incoming mail items must be directed to the sorting object. In general, a mechanism is needed to forward data arriving on an input link of a composite object to an input link one of its components. This is achieved by allowing component objects to access the input link of the composite as though it were an output link. Within the specification, the composite is referred to as self. Constraint 1 uses this mechanism and the built-in predicate Linked() to ensure that the external input link of the composite object is connected to the input link of the mail sorting object.

Constraint 2 indicates that the output link of the buffer object connects to the input link of the sorter. When the buffer is flushed, mail items are re-directed to the sorter and re-processed. Constraints 3 and 4 ensure that the outputs of the sorter are always connected to something. Constraints 5 and 6 refine this requirement by demanding that each output link be connected to either a mail transport object appropriate for that link, or to the buffer. Finally, constraints 7 and 8 guarantee that the only time an output link is connected to the buffer is when there is no suitable mail transport object available. The combination of these various constraints ensures that mail items are always directed to a suitable transport service object if one is available. Otherwise, they are directed to the buffer object.

As noted in section 2.2, the MHS composite uses a script-based repair mechanism. The repair clause of the MHS composite consists of a series of imperative programs. Each numbered constraint of the specification clause has a corresponding repair script in the repair clause. When a given constraint fails, its associated repair script is executed, and the composite is re-verified. Repair scripts 1 through 6 all carry out simple link connections. System is a predefined object which provides access to various features of the underlying runtime support system. The link operation, as might be expected, establishes a connection between the provided object/output-link and object/input-link.

Repair scripts 7 and 8 are slightly more interesting. The two scripts are quite similar, one re-linking the low priority output of the sorter and the other the high priority output. The first statement in each uses a set constructor to obtain a set of the available mail transport objects based on priority. The method getAny() is then used to randomly extract one of the mail transport objects from this set. A connection between the sorter output and the chosen mail transport object is established. Referring back to constraints 7 and 8 of the specifications clause, it is easy to see that if either of these constraints fail, then the set of suitable mail transport objects cannot be empty, hence, it is unnecessary to check for this condition in the repair scripts. Finally, having re-established a connection to a transport object, the flush() method of the buffer object is invoked, causing buffered mail items to be re-sorted and possibly delivered via the newly available transport service.

In addition to the different repair mechanisms used, the directory tree and mail handling system examples differ in the manner in which RCMS is used. One can regard the directory tree as part of a pre-existing service which is subsequently monitored and repaired by RCMS. In the mail handling system, on the other hand, RCMS is used to construct a new system from existing component types. Use of RCMS with an existing service is facilitated by parameterization of a composite via its constructor and public instance variables. For example, an entire directory tree can be placed under RCMS monitoring simply by passing off its root to the constructor for DirTree. Because public instance variables are used, the full structure of the directory tree is still available to its users. Use of the RCMS to construct new systems is facilitated by the use of inter-object links and private instance variables. In this case, the internal structure of the composite is hidden and the service is available only through the defined interface of the composite object.

3. Implementation of RCMS

This section provides a brief overview of how RCMS is implemented and highlights some of the unique problems encountered in attempting to provide a powerful high-level specification language which nonetheless demonstrates acceptable performance.

3.1. Current status

The implementation of RCMS consists of:

1. A compiler (rcfg) which translates composite object definitions into Raven code.
2. A set of Raven class definitions and implementations, which are used by rcfg-generated code.
3. A set of run-time procedures providing services for object monitoring and for tracking membership in composite objects.
4. Modifications to the transaction manager of Raven to allow verification of a composite object prior to committing a transaction.

As noted in (1), composite object definitions are compiled to Raven code. Hence, the composite objects are represented within the Raven system as a collection of elementary Raven objects. For each composite object definition, rcfg creates a corresponding Raven class definition which describes the composite. The class definitions for all composite objects are descendents (direct or indirect) of the class Composite. This class defines the minimal interface for a composite object and provides implementations for general purpose configuration management tasks. Each composite object maintains a parts list of the components of the composite object it represents.
In addition to generating an interface description for the placeholder object corresponding to a given composite, rxfg also generates code for several behaviours: the constructor, a behaviour for updating the parts list (updateParts()), and a behaviour for computing whether the composite satisfies its specification (verify()). Figure 5 is a listing of the output of the RCMS compiler for the directory tree composite. It has been edited slightly to improve its readability. The keyword recoverable on the class declaration for DirTree indicates that atomic recovery is to be used for objects of this class.

The constructor for the composite takes as parameters those specified in the constructor of the composite declaration. These parameters are then assigned to instance variables of the composite so that they are accessible to the code generated for the specifications clause. It then calls the updateParts() behaviour to cause the initial parts list to be computed.

The behaviour updateParts() ensures that the parts list for the composite is consistent with the current state of the composite. It is computed syntactically from the contents of all instance variables which have the attribute indirect. The only manner in which the parts set may change is if an instance
variable is assigned a value via the := operator. The basic task of updateParts() is to compute current values for these instance variables and update the parts set appropriately. The only interesting task here is the recomputation of set constructors (expressions of the form all X : logical expression). Such a set constructor is used, for example, to define the instance variable nodes in the directory tree of figure 1. The domain of quantification for these set constructors is the change set of the composite object. The change set is the transitive closure of the 'refers to' relation, starting from the instance variables of the composite object. An object A is said to 'refer to' another object B if an instance variable of A contains a capability for B. Each set constructor is evaluated by computing the change set, then iterating over it and evaluating the membership predicate for the set.

Use of the change set as the domain for evaluating set constructors represents a trade-off between expressiveness and efficiency in RCMS. Using the change set restricts the sorts of predicates which can be used to define the components of a composite. For example, suppose that one wanted the expression all X : X.Size > 1000 to generate the set of all objects in the system whose size was larger than 1000. Implementing these semantics requires monitoring every object in the system so as to detect whether its size falls above or below 1000. Obviously, the creation of a composite whose parts list requires the evaluation of such an expression is not reasonable.

If the domain of quantification is restricted to the change set, the loss of expressiveness is not great. All predicates which are expressed in terms of object references will evaluate correctly under this scheme. For example, the Path0() predicate used in the directory tree is based on the in operator, which in turn relies upon the existence of object references from the contained to the containing object. The use of transitive closure in the creation of set constructors does not mean that sets such as that defined above cannot be used at all in RCMS. It is perfectly reasonable to monitor the sizes of the components of a composite object. However, it is not reasonable to attempt to create a composite based on such criteria.

In many cases the computation of the change set can be avoided and the value of a set constructor computed directly. Returning again to the directory tree, it is clear that the most efficient way to compute the value of nodes is a simple depth-first recursion of the directory tree using a marking scheme to avoid cycles. Evaluation of such sets has been an ongoing concern for database researchers, and methods have been developed to translate predicate-based set constructors into efficient procedures for computing their value [2]. Any membership predicate which is linearly recursive can be translated into an efficient set generator. The definition of linear recursive is somewhat technical, but it encompasses a broad class of predicates including those such as Path() as used in the directory tree example.

Optimizations such as this are important for RCMS because maintaining a correct parts list for each composite is fundamental to the operation of RCMS. Only by knowing what objects are members of each composite can RCMS effectively monitor changes in the system and determine when a change in the state of the system may have affected a particular composite. RCMS maintains the relationship between composites and their components by storing in each object a list of references to the composite objects of which it is a part. If an object changes state, it is thus a simple matter to determine which composite objects have been affected and require re-verification. These lists of references in each object are kept up to date by deriving them from the part lists associated with each composite.

These lists of references, when combined with a unique feature of the Raven system, provide the basic monitoring mechanism in RCMS. Raven allows special handler routines to be attached to the basic invocation mechanism which lies at the core of all object interaction in the system. Such a handler is called before or after the actual execution of the method code during the invocation process. A particular handler is only called if the object being invoked upon has a flag set indicating that the handler should be used. RCMS implements monitoring via a handler routine which is called after the execution of the method code. This handler checks to see if the object has changed state. If the object has changed state, then RCMS determines which composites the object has been tagged with and causes re-verification of those composites.

Monitoring for composites which use atomic repair uses a somewhat different mechanism. In effect, object monitoring is handled by the transaction manager, since it must track all objects which have a given transaction has touched. When a transaction commits, all of the objects which have been touched by that transaction are checked to see if they belong to any composite objects. If so, then RCMS forces re-verification of those composites to occur. If any of the composites is not in a valid state, then RCMS aborts the transaction.

The Verify() behaviour is responsible for checking whether the current state of a composite object satisfies its specification. The evaluation mechanism is straightforward because the domain in which the constraints are evaluated is finite and quantification over integers is not permitted. Essentially, a quantified expression is compiled into a loop which iterates over the parts lists, binds the iterated values to the quantified variable, and then evaluates the body of the quantified expression.

As noted above, verification failure for a composite using atomic repair simply aborts the transaction which leads to the violation of the composite's specification. Thus, the composite is returned to its previous valid state.

When atomic repair is not used, verification failure causes a repair script to be run. The repair script is chosen on the basis of which constraint of the specification clause failed. Once the repair script has been executed, the composite is again re-verified. This cycle of verification and repair is repeated until the composite reaches a valid state or a maximum number
of iterations is exceeded. In the latter case, the normal operation of the composite object is suspended and the system operator is informed of the problem.

3.2. Optimizations

A variety of optimizations in object monitoring and composite verification is possible. A number of these optimizations are described elsewhere [4], but it is worthwhile to consider at least one of them in the context of this paper: evaluation of quantified expressions. Notice in figure 1 that the quantified expression contains a term enclosed in brackets. This term is known as a filter expression because it filters the values which the quantified variable takes on. Strictly speaking, filter expressions are not necessary because they can be merged with the body of the quantified expression through conjunction. They exist within RCMS in order to improve readability and to make optimizations easier to carry out.

Iteration over the complete change set in order to evaluate quantified expressions is frequently not required. Referring again to the example of figure 1, if symbol tables have a mechanism for iteration over the set of values they contain, then the filter expression Z in X means that we can evaluate the quantified expression by iterating over the values contained in X rather than the parts set. The current version of rcfg recognizes a class of filter expressions for which this optimization is possible and emits the corresponding code.

4. Comparison with other systems

RCMS explores territory which is being actively investigated within the context of other projects such as the Network Management System (NMS) [17] and Conic [12]. From the outset, the goal for RCMS has been to provide a general-purpose, automatic, and distributed approach to configuration management. Other configuration managers provide an environment which is quite distinct from the underlying system in which the components of systems are programmed. For example, the NMS uses an active database to maintain configuration information. Automated re-configuration of a system is governed by rules for the database. Conic provides a separate imperative configuration language and provides only very limited support for automated re-configuration.

RCMS extends an existing object-oriented language in a manner conducive to configuration management tasks. The structuring techniques used in object-oriented programming are thus available to aid in configuration management. The impedance mismatch between the configuration system and the imperative programming system is minimized.

RCMS directly associates configuration management information with the managed objects. Distribution of configuration management activities is thus the natural outcome of the distribution of the objects themselves. In NMS a similar degree of distribution would entail the additional overhead of moving from a centralized database to a distributed one. The MAD [6] approach to network management provides a mechanism for distributing management activities, but it lacks the simplicity of RCMS because the distribution of management activities is not integrated with the distribution mechanisms of the underlying system.

Configuration information in RCMS is expressed declaratively as a sequence of constraints which delineate valid states for the composite object. The declarative form is naturally suited to the task, as configuration management generally involves ensuring that a system operates within particular boundaries. Furthermore, specification of what a composite object should look like is separated from the description of how to achieve it. This distinction is important because it keeps the specification of valid states uncluttered and permits independent re-use of specifications and repair techniques. NMS also uses a declarative form, a set of active database rules. Although declarative, these rules are still somewhat in the vein of specifying how to carry out configuration management tasks. Conic provides an imperative language for carrying out configuration tasks but, as noted above, does not provide the facility for automating these tasks.

5. Conclusions

RCMS provides a new approach to configuration management. Its seamless integration with the object model of the underlying system, its declarative language for specifying configuration information, and its automated re-configuration mechanism combine to produce a flexible and powerful solution to the challenges of configuration management. The current implementation of RCMS demonstrates feasibility. Clearly, improvements in efficiency are needed. Fortunately, despite the general nature of RCMS, numerous possibilities for optimization are apparent and many more will undoubtedly be discovered as research progresses.

Object monitoring also offers avenues for large efficiency gains. The current scheme offers a very coarse level of monitoring, simply detecting whether an object's state has changed. Suppose, however, that one of the constraints for a composite is foo.priority < 3. Then, the invocation routine for foo could be replaced by one which specifically monitors for this condition. The knowledge that this particular constraint has been violated might allow further optimization of the verification behaviour and the repair system.

Finally, both the repair mechanisms and the specification language may require extensions. The NMS system [17], for example, supports a query language which permits the definition of 'events' defined in terms of particular clock times, values of attributes, or durations of other events. The utility of these features within the NMS suggests that they should be explored in the context of RCMS. Work is also being directed
towards developing advanced repair techniques. For example, the specification for composite MHS (figure 2) contains sufficient information that a constraint solver could determine which links needed to be established or broken, without any imperative directions from the programmer.

Overall, the development of RCMS has been most satisfactory. It demonstrates the feasibility of general-purpose configuration management in an object-oriented environment. A declarative language for expressing configuration information is both concise and intuitive. The implementation of the system supports a wide variety of optimizations, yielding performance which will allow practical application of RCMS.

References


[15] Ploedereder E and Fergany A 1989 The data model of the configuration management assistant ACM SigSoft 17 (7) 5-14
