Towards a Requirements-based Information Model for Configuration Management

Gerd Aschemann Roger Kehr
Department of Computer Science
Darmstadt University of Technology, Germany
{aschemann,kehr}@informatik.tu-darmstadt.de

Abstract

Several architectures are defined for distributed systems management, most of which stem from the network management domain. They all share the idea of multiple models, at least comprising an information model, a communication model, and a functional model. Configuration management is part of the functional model but is very dependent on the information model. Due to its history in network management, the information model itself is mostly restricted to describe a priori known properties of hardware entities or software entities closely bound to the hardware point of view. Modeling the dynamic relationships of distributed systems and applications with such a model is hard if not impossible. Therefore we propose a new information model that focuses on the dynamic description of distributed systems. By using object-oriented technologies such as a prototype-instance model, it is well suited for the actual demands of system administration. Since we do not explicitly combine it with a particular distribution mechanism, it is orthogonal to any given or future communication model. We present an exemplary case study in the analysis to our approach, and we describe the design and implementation of our model.

Keywords: *distributed systems, configuration management, value sharing, inheritance, information model, consistency specification, constraints*

1. Introduction

Traditionally, configuration management has been acknowledged as one of the main functional areas in the management of distributed systems. Together with fault management, accounting management, performance management, and security management (in short FCAPS) it forms the so-called *functional model* of the OSI management architecture [15]. This architecture stems from classical network management but also serves as a reference model to both, other (network) management architectures (e.g., SNMP, TMN, WEBM, JMAPI) and higher level management architectures for the management of distributed systems and distributed applications [7]. Besides the functional model, these architectures usually also define other models, comprising at least an *information model* and a *communication model*.

The information model describes the properties of managed systems relevant to management. Therefore it contains the information to configure these systems as well as a substantial amount of other information that is relevant for the other functional areas. Since it originates from network management, the information model is well suited for a priori known attributes of isolated entities, implemented in hardware or firmware. It mostly has a class-based or hierarchical structure and therefore lacks possibilities of dynamically modeling properties and their relationships in a distributed system.

The configuration view on a managed network focuses on the given equipment and its possibilities: It is technology driven. Up to now, the typical starting point to configuration management is the question "what can we do?" Due to the growing complexity of distributed systems and applications, this target moves to a more requirements-based approach, i.e., the question "what do we want or need to do?" System administrators are more and more faced with the requirement to maintain a complex network of related services and to distribute them on the given heterogeneous hardware, featuring advanced techniques like aggregation, migration, replication, consistent reconfiguration, etc.

We propose a new information model for configuration management. We leave the classical class-based and therefore static information models and introduce a prototype-based approach which allows to concentrate on the relevant properties to compose a distributed network of service providers and customers. By using inheritance it is easily possible to describe large groups of objects which share

ICCDS IV, May 4-6, 1998, Annapolis, MD, USA

the same properties and only to define exceptions and extensions where necessary. It allows to avoid redundancy which is often found in configuration management. Our model provides a powerful query and constraint definition language to define sets of objects and predicates for properties and relationships. It is thereby adequate to the well known concept of domains [16] and policies [12]. We believe that an appropriate modeling of a distributed systems configuration is a good basis for management in the other functional areas like fault or security management. As an example, if a server crashes, it can be simply figured out which services are directly and indirectly concerned, and appropriate reconfiguration measures can easily be deduced within the model. This allows for a fast and complete recovery from the faulty situation.

Unlike other architectures, we explicitly do not propose a communication model. Our information model may be integrated with various communication models:

- It can act as the source for regular information and directory services (e.g., regular files, DNS, NIS, X.500, LDAP) with their distribution models.
- It can be the source for configuration management in the existing open or proprietary architectures.
- Last but not least one can build a new communication facility to apply the configuration to the managed systems. This new facility can be based on both paradigms, a push or a pull style or a combination of them.

The next section gives a detailed and systematic analysis of the problem in the form of a case study. Section 3 shows the design of the model, while Section 4 describes its implementation and Section 5 introduces a short example of the model. An overview on related work is given in Section 6. Before we conclude in Section 8, we briefly introduce some currently investigated extensions to the model (Section 7).

2. Analysis

Consider the following "NFS configuration" scenario [18] which is familiar or at least easy to imagine for system administrators and typical for other patterns in configuration management. In a grown environment like a research institute or a software development department, we often find many more or less equally equipped workstations cooperating in a peer to peer manner (i.e., in NFS configuration every diskfull system serves its disks to the other systems). There are no or only few dedicated file servers. All systems act as clients to more or less all other systems. Mostly disks are mounted either statically at boot time or dynamically by an auto mounter. The latter eases reconfiguration but does not automate it. System administration is faced with the following configuration issues.

2.1. Configuration by Policy

Policies System administrators or business officers want to describe system administration policies in an abstract way. This may be rules like "every host with sufficient disk space should hold and export home directories" or "every subnet must contain an application software server and all clients in this subnet should use this server as primary resource". On the other hand, there could be restrictions like "disks holding sensitive data must not be exported to another department".

Policies can be seen as more politics-based constraints to distinguish them from the more technical-based consistency rules.

Consistency We use the term consistent configuration if the configuration holds against a technical constraint, e.g., "a file system shall be mounted or even be included into an auto mounter configuration only if it is exported to the appropriate clients". Another constraint could be that "a client set must not be empty" which could easily occur if sets are described in an abstract way.

2.2. Maintainability

Like in software engineering, maintainability is much easier if built into the specification part of the system rather than into the implementation or application of the system. Policies and consistency rules are specifications in a descriptive sense. Furthermore, maintainability needs support of the following techniques.

Abstraction Several levels of abstraction should be possible—policies are mostly at the highest level. Suppose an application software server exports a disk with platform specific software, i.e., software for a specific combination of processor and operating system such as `sparc-sunos` vs. `sparc-solaris`. The set of clients should be defined by a description like "all hosts belonging to that platform" instead of a concrete list of hosts that would be hard to maintain.

Reuseability All equally equipped systems share their disks under mostly the same conditions to their clients. Only few of them are bound to exceptions like restrictions or extensions to client access rights or have other extensions like additional disks. There should be a way to describe the default behavior as a prototype and allowing for the description of exceptions and extensions. These extensions and exceptions should themselves be subject to prototyping, if appropriate.

Extensibility Prototyping also eases extensibility. For example, adding new hosts usually means to check the configuration of every service they offer. The vendor-supplied configuration often does not match the system administrators requirements. If the new system is comparable to existing ones, it should be easy to apply the same configuration pattern to it and only check for exceptions and extensions.

Changeability After moving a service (e.g., data to another server), easy reconfiguration should be supported which preserves all predicates defined for the configuration parameters belonging to the service.

2.3. Application to the Real World

Mapping more or less abstract configuration descriptions to the real world has to cope with additional problems.

Heterogeneity Different operating system vendors often describe the same aspects in rather different ways. For example, some require a set of permitted clients to be a list of (canonical) host names, while others allow for wild-cards or predefined sets of names like NIS netgroups. On the one hand it must be possible to describe such information in an abstract, extensible, and vendor-independent way. On the other hand transformations from the abstract description to the vendor-specific interface must be provided, be it a file or a repository with its own syntax, a command-based interface, or something else. Even if a new system or a group of new systems extends heterogeneity of the whole zoo, the policies usually remain the same. In this case, a new transformation of abstract descriptions to a concrete interface must be integrated.

Replication of Services Important services, e.g., application software in our scenario, should be available multiple times to increase reliability. If the servers are imported by an appropriate auto mounter configuration, mostly automatic recovery from a file server crash can be achieved on the client side.

Aggregation of Services Sometimes different services are composed to form a new higher-level service. For instance, boot service for diskless systems like X-terminals, network computers, network printers, or nomadic systems like laptops often require BOOTP or DHCP besides NFS. A technical requirement (a consistency constraint) is to perform such a service at least in the same network segment as the client systems if not from the same server.

3. Design

3.1. Object-oriented Configuration Model

Uniform representation of data is one key component that makes reasoning about information possible. The first step towards a uniform representation of information is to use *objects* as the central modeling entity. Object-oriented modeling in the domain of software development is a defacto standard. Applying object-oriented techniques to configuration management is the central shift in our approach that offers several new aspects in configuration systems.

Representation Objects allow for the abstract representation of information without considering the concrete syntax in a particular environment. *Transformers* that generate the *concrete syntax* can be used as back-ends for configuration systems. Using more abstract representations yields flexibility in configuration processes and re-usability of configuration information.

Aggregation Aggregation is one of the fundamental principles of object-oriented modeling. Objects can be composed to form larger *aggregate objects* to obtain new configuration entities with new properties. This fundamental modeling technique should be available in configuration systems.

Relations Objects in the real world are usually related to other objects. For example, they describe relations such as *depends-on*, *is-a*, *part-of*, etc. In configuration systems most objects are not isolated and are in relation to other objects. A configuration system should allow the management of these links in appropriate ways. They could be used to reason about the relations in a system. Making these links explicit is a form of *documentation* that may be necessary to describe a system. In particular *inter-service* dependencies (e.g., a service which can only work if another service is up and running) cannot be expressed in most configuration systems.

Query Language Since the representation of objects follows a fixed schema, query languages can be developed to query all kinds of information in a uniform way. This is particularly important in heterogeneous systems where uniform queries are hard to implement. Queries are often used to describe a high-level policy such as “all hosts in that building use that primary name server”. The ability to store queries itself as first-class persistent objects makes it possible to represent such queries declaratively within a system.

3.2. Repository

In Figure 1 the overall architecture of our system is shown. A central repository manages all kind of configuration information. Objects are stored in the repository and can be transformed in to a concrete syntax as necessary. The transformation step can occur inside or outside the repository yielding so-called *external documents*. Further distribution of documents can be done using reliable transport systems.

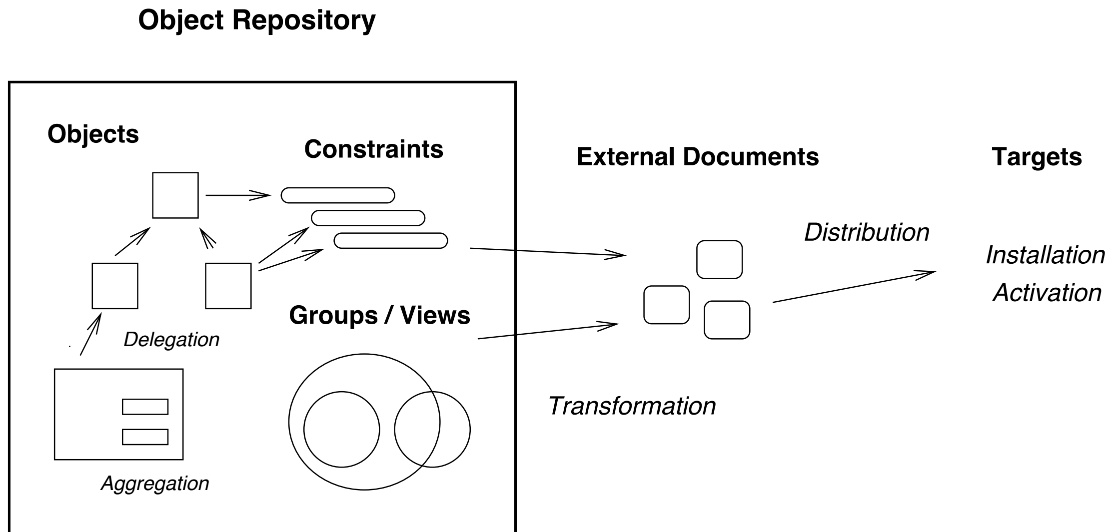

Figure 1. Object Repository Embedded into a Larger System Architecture

3.3. Object Model

Objects are the fundamental concept of organization in our system. An object has an *identity* and consists of a set of *slots*. A slot is a tuple $S = (name, value, attributes)$. The values of slots can be primitive types such as numbers, strings, etc., objects, and links to other objects. Slot values can be *evaluated*. Objects can be stored as slot values which allows for the creation of aggregate objects. An object can be a part of at most one object. *Links* are slot values that reference other objects. Thereby, arbitrary graph structures can be constructed.

Each object has a unique *name*. Objects are organized in a *hierarchical name space*. The name space can be compared to hierarchical name spaces in traditional file systems. Objects can be seen as a *directory* containing other objects.

3.4. Inheritance by Delegation

As previously described, redundancy of configuration information should be avoided in configuration systems. One common way of reusing information is *inheritance* [19]. Typically, inheritance is applied in programming languages to achieve *code reuse* whereas reuse of values is of less concern. The research on *prototype-based* programming [11, 21] has proposed to integrate *value sharing* among objects into programming languages. In this model objects are not an instance of a particular class. Therefore the notion of *class-less* or *object-centered* programming are often used to describe this kind of concept.

As noted in [10], strictly class-based modeling can have a significant impact on the modeling flexibility which conforms to our personal experience in the domain of system and network management. Therefore, we have decided to follow a more dynamic approach where objects can be created on-the-fly and whose structure can be modified without any prior class-like description. One may be tempted to assume that missing structural definitions may lead to an unmanageable object pool. As we will see later, this *schemaless* modeling is not a disadvantage at all since *constraints* can be used to achieve the same results in a more flexible way.

A special relation called *delegation* [11] can be used to implement inheritance among objects. The values of slots are requested by sending an appropriate message to an object. A message is a tuple $M = (object, slot\text{-}name, arguments)$. If *object* has a slot named *slot-name*, the value of the slot is returned. If the object does not own a slot with this name it *delegates* the message to its *parent object*. A special relation called *parent-of*-relation links objects to their parents. Hence, the slot look-up succeeds if at least one parent can return an answer to the original message.

This principle eases the description of *default objects* that implement *default values* for configuration entities. New objects can be connected to these objects with the parent-of-link resulting in an inheritance of all default values. Local modifications to the new object can be done to describe *exceptions* and *extensions* from the default object.

3.5. Integrity Checks with Constraints

The second important aspect of our model is to offer concepts for the specification of *syntactic* as well as *semantic integrity checks* on a system configuration. One advantage

of representing all modeled entities as objects is that all relevant information is uniformly accessible in the system. It allows to reason about a configuration and to detect modeling inconsistencies prior to installation and activation of the configuration information.

To be as flexible as possible, our system allows for the attachment of arbitrary *predicates* to objects that implement system contstraints. These predicates are written in COMMON LISP [17] and have free access to the objects inside the repository. Since predicates are completely separate from each other, each predicate declaratively *guards* an essential property of the system. We have chosen LISP primarily for its richness in high-level abstraction principles, its ability to add new functions at run-time in an elegant way, and being reasonably fast even for complex integrity checks. Other scripting languages—such as TCL, PYTHON, or SCHEME—could have been chosen as well.

Currently, integrity checks can be triggered manually that evaluate all predicates in a system and issue appropriate warnings if conflicts or inconsistencies arise. This scheme can be improved with the use of data dependency management as implemented in the KR system [5, 13]. Modified data can then be used to trigger all dependent contraints as well, leading to an incremental constraint checking scheme and thus minimizing re-computation costs.

3.6. Computed Slots

Slot values can also be LISP functions taking arbitrary arguments. *Computed slots* are parameter-less functions that have access to the whole system. They are stored as ordinary slot values and can be used to implement *behavior* in the system, such as the transformation of objects into a concrete syntax for a certain application. Additionally, certain operations in the repository can be automated by LISP functions that have side-effects on the repository. Hence, the system can be characterized as a repository that focuses on the management of data but additionally offers full programmability.

Similar to incremental constraint checking, computed slots can take advantage of data dependency management leading to a scheme that allows the inference of all dependent values if changes in the repository have been performed. This enables the redistribution of only those data that actually have been changed. Based on our experience with todays constraint-solver technology, we have designed our object model with strong consideration of a future integration of a solver to our system.

With computed slots and queries one can represent *object groups* as *first-class* objects. A *group* can be seen as a persistent set or list of objects obtained by a query. This is similar to *views* in database management systems. Groups can be constructed from other groups by using arbitrary set operations such as *union*, *intersection*, or *set difference*. We believe that such high-level descriptions of object groups are a very important design concept underestimated in current system configuration environments.

4. Implementation

The current implementation of our configuration system consists of several parts.

Repository The current system consists of a repository implemented in LISP. It implements the object model and the interaction to backing store. It is connected to an TCP/IP socket that offers full access to the repository. *Persistence* is currently achieved using a load/store model.

Web-Interface We have implemented a web-interface based on CGI-scripts that connect to the repository via the socket interface. The contents of the repository may be browsed and objects may be inspected. Links in the object model are represented as hyper links. Local and inherited slots may be visited easily. An online demo of the interface may be found in [9].

Corba-Interface A CORBA server [14] has been implemented that makes the repository accessible from arbitrary CORBA clients. The server is connected to the repository and simply transforms incoming method calls into appropriate requests for the repository.

5. Example

Giving a complete introduction into syntax and expressiveness of our approach is beyond the scope of this article. Figure 2 shows a brief and syntactically incomplete example of a system configuration, cf. [8] for further details.

Basic Mechanisms At first we take a look on the objects and the groups with the solid lines on the left side and the center of the figure. It describes a network of Sun workstations with NFS servers and clients. Note that some object names, e.g., of objects *A* to *F* start with the word :proto. This is only a naming convention, we use these objects as prototypical descriptions for other objects, but they need not be named with the :proto prefix. The first two objects (*A* and *B*) have simple slots with text and integer values. They are used to describe the CPU type and operating system property of host objects. CPU and OS together form a platform, object *C*, which holds references to the former simple objects. A prototypical host object (*D*) is simply characterized by its platform. A server host prototype (*E*) for the same platform is derived from this host prototype, thereby inheriting the platform information, and extended by the slot EXPORTS. Object *E* now has two slots, PLATFORM and EXPORTS.

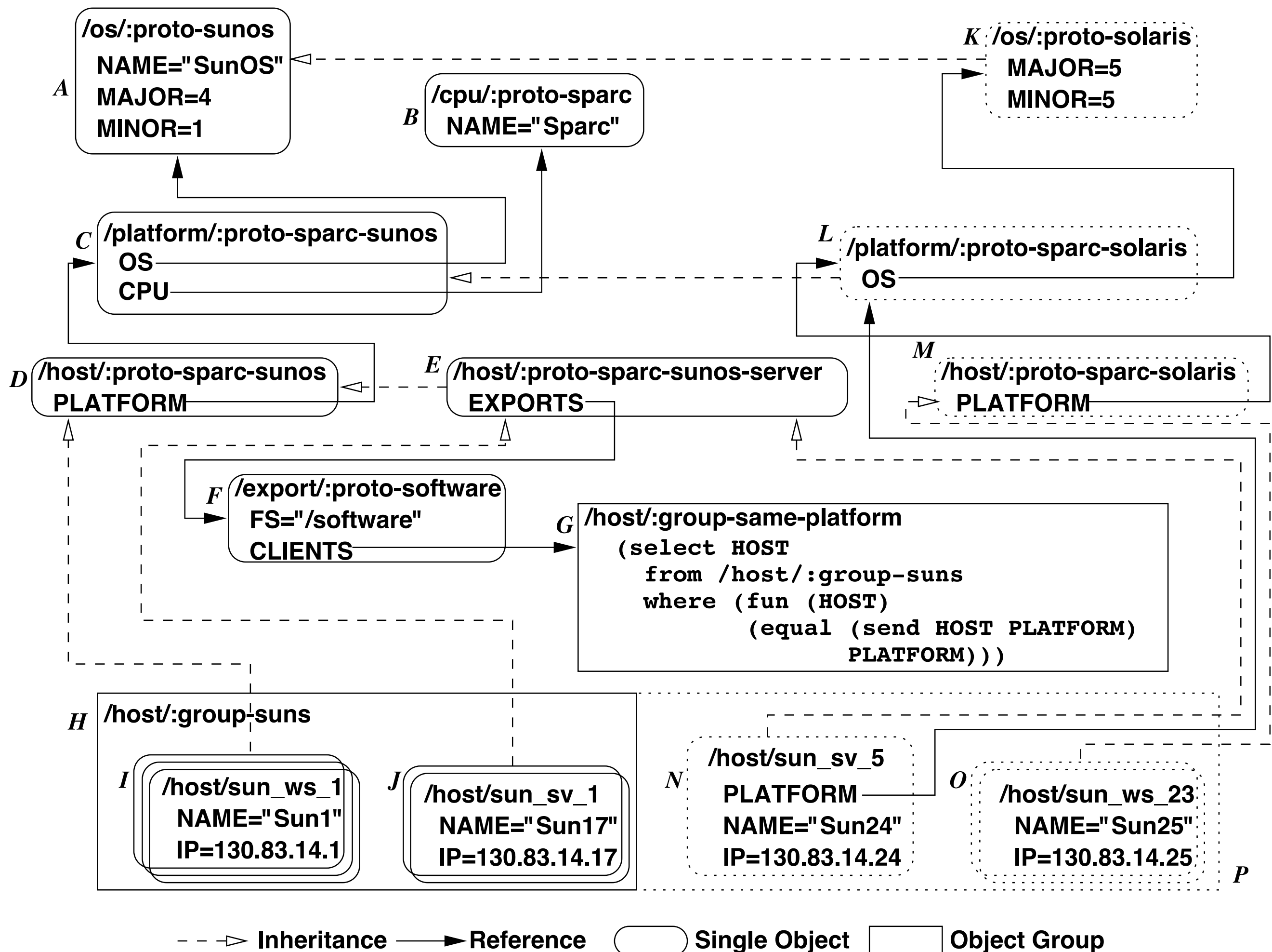

Figure 2. Configuration Example

Query Language The exports object *F* gives a file system (slot FS) and a list of clients *G*. This list is a very simple example of the query language. It selects all hosts from the group of Sun hosts (/host/:group-suns) with the same platform as the host having the EXPORTS slot. It can be used to compute the list of NFS clients of a host. In real systems management we would not directly compute the list of NFS clients for the /etc/exports file of each host but use NIS and generate appropriate netgroups instead.

Real World Objects Now we can start to define the first representatives of real hosts. Some workstation objects (*I*) are directly derived from the platform prototype (*D*). Each of them has its own name and IP address. In the same manner we define some server hosts (*J*) by inheriting from the server prototype *E*. In fact they now have four slots, PLATFORM, EXPORTS, NAME, and IP. All hosts together (*I* and *J*) form the group of Sun hosts (*H*).

Extension Now we want to extend our world of Sparc-Sunos hosts by a new operating system. This could be the beginning of a migration process to the new OS. Therefore we introduce the dotted objects on the right hand side of the figure. We derive a new OS prototype (*K*) from the Sunos prototype. The OS name remains the same but major and minor release number are overwritten. The same happens by the new platform prototype (*L*) which inherits the CPU information from *C* but overwrites the OS slot. Consequently we introduce a new host prototype for the new platform (object *M*). Note that we do not set up a new server prototype like the object *E* for the new platform. In the beginning we start with only one server object (*N*) and do not need a server prototype. It can be easily introduced on demand if the migration process requires more servers. Our only Solaris server inherits from the given server prototype (*E*) but overwrites the PLATFORM slot by the appropriate reference (to object *L*). The latter is necessary, since we do

not provide multiple inheritance which has difficult semantics in general, e.g. which inherited information is valid if we multiply inherit the same slot. The list of clients will therefore be correctly computed by the given select expression G although it was originally specified for another platform. Finally we have some Solaris workstations (O). Migrating hosts from, e.g., J to O is simple by changing the inheritance relation to the new host prototype M. All new (or migrated) hosts N and O form the extension P of the group of Sun hosts H.

NFS Imports However, in the figure we explicitly describe only the NFS export relationships. In the real world we would have additional information about the hosts, e.g., their location. Since we have — at least in the Sunos case — redundant servers, there could be a computed slot SERVER for every client, referring to the closest server. Together with an auto mounter we can also take advantage of the redundancy by computing also one or more backup servers for the clients. In case of a server fault, the auto mounter would automatically mount a backup server.

Transformation and Installation We also do not show the transformation of internal information to external representations. As we mentioned above there are several possibilities. Currently the repository is a single central instance. Since it is written in LISP we can directly generate files like NFS export lists, mount tables etc. These files can be applied to the different hosts by well known communication facilities (like the UNIX tools rcp, rdist or rsync). Some of the information will be put into appropriate naming services like NIS or DNS and distributed by their specific mechanisms. A very other possibility is to run special agents (e.g., UNIX daemons) on every host and let them replace the changed information in the local configuration repositories, e.g., plain files or registries. In all cases, however, we can take advantage of a dataflow analysis of the changed information and regenerate and replace (reconfigure) only the affected external configuration files and local repositories on the affected hosts.

6. Related Work

Many configuration systems combine an information model with the distribution and installation models. Since we are primarily interested in the information model, we concentrate on the information models of these systems.

CFENGINE [2] is a configuration system that combines an information model consisting of *patterns* and associated *actions* with an *interpreter* that performs the actions if patterns match. Patterns are matched against an interpreter's current context—such as host name or operating system version—and in case of a match the corresponding actions are executed, typically resulting in the change in the configuration of a service on the host the interpreter is currently running. Due to the potentially unknown context the declarative description is applied to, it is inherently difficult to implement integrity checks, since no *extensional* description is available.

GENUADMIN [6] implements a simple inheritance mechanism among so-called *stanzas* that describe configuration entities as its basic abstraction principle. Stanzas exist in a flat name space. Higher-level abstractions such as sets and set operations or computed slots as well as integrity checks can only be realized as external extensions.

LCFG [1] introduces a three-layered name space of the form (*host, subsystem, attribute*) to store information. The installation model is separate for each subsystem. Similar to GENUADMIN, other services must be implemented externally to the information model.

The Raven Configuration Management System [3] combines a class-based object model with configuration assertions written in a first-order calculus language. It supports mechanisms for automatic reconfiguration by applying monitoring techniques, and similar to our approach, focuses on the integration of managed objects and configuration information. As far as we know, information sharing among objects and schema changes at run-time are not directly supported.

In [20], basic configuration data is managed by relational tables, whereas modular entities called *prescriptions* are used as central specification components. They implement a hybrid approach that combines *logic-oriented* specifications used for verification with an *operational* processing model for the realization of repair steps of configurations. In contrast to our work, the main focus lies on the integration of descriptions with repair, whereas our work currently focuses on abstract representations of the description of the basic configuration information combined with predicates that may have side-effects on the contents of a repository.

7. Future Work

Several extensions of the current system are planned.

Improved Web-Access We plan to extend the web-interface to offer easy access to the repository. The main focus lies on a user interface based on an appropriate interaction model with the repository.

NFS-Server Since the name space is organized similar to a hierarchical file system, we plan to implement an NFS-server that maps file system operations onto the repository. Since the semantics of a file system are different from the repository semantics, we are not yet sure to what extend a useful integration of both concepts is achievable.

Constraints As outlined in Section 3.5, we plan to add a constraint-solver into the repository that manages all data

dependencies among objects, predicates, and computed slots. This enables the automatic detection of all depending objects in the repository that are affected after a value change has occurred. This should result in minimized distribution costs and add monitoring capabilities to the system.

Modeling Patterns Similar to design patterns [4] we believe that more modeling patterns may emerge in a particular configuration domain which we believe is a potential area of research. Our approach offers the potential extensibility of the repository to represent such patterns both on the language and the object level.

8. Conclusions

As we have seen in the analysis of the requirements for configuration management, there is a particular need for a new information model. It should be focused on the demands in configuration management instead of the very detailed description of hardware and software properties which are mostly only useful to other areas of management, i.e. failure, security, or performance management. Besides this we believe that management of these other areas can be better supported if the appropriate information models were based on well-suited configuration descriptions.

Finally, we have shown that the given problems in Section 2 can be solved with the proposed model given in Section 3. Our prototype-based object model allows for description of dynamic relationships, abstraction, and extensibility. By inheritance of data and groups we provide reuseability and changeability. Policy management, consistency, and integrity checks are integrated by a powerful query language, predicates, and computed slots. The latter also provides a mapping of abstract information to service configurations in the managed systems with respect to their heterogeneous syntaxes. The required aggregation of services can directly be expressed by aggregation of objects. Finally, the description of the redundancy of services is supported by sharing information among objects without the need for redundancy in the object repository.

References

[1] P. Anderson. Towards a High-Level Machine Configuration System. In *Proceedings of the 8th USENIX Conference on Large Installation System Administration (LISA)*, pages 19–26. University of Edinburgh, Sept. 1994.

[2] M. Burgess. *Manual of cfengine version 1.3.6*. Centre of Science and Technology, Faculty of Engineering, Oslo College, Norway, June 1996.

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

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts, 1994.

[5] D. Giuse. *KR: Constraint-Based Knowledge Representation*. Carnegie Mellon University, Oct. 1993. This document is part of the GARNET [13] distribution.

[6] M. Harlander. Central System Administration in a Heterogeneous Unix Environment: GeNUAdmin. In *Proceedings of the 8th USENIX Conference on Large Installation System Administration (LISA)*, pages 1–10, Sept. 1994.

[7] H.-G. Hegering and S. Abeck. *Integrated Network and System Management*. Addison-Wesley Publishing Company, Reading, Mass., 1994.

[8] R. Kehr. SCOT — System Configuration Tool, Entwurf und Implementierung eines Frameworks zur Realisierung von Informationsmodellen für Konfigurationssysteme. Master's thesis, FG Verteilte Systeme, FB Informatik, TU Darmstadt, Sept. 1997.

[9] R. Kehr. The Scot Home Page. *http://www.informatik.tu-darmstadt.de/VS/Forschung/Management/SCOT/*, Oct. 1997.

[10] B. Krogh, S. Levy, A. Dutoit, and E. Subrahmanian. Strictly Class-Based Modeling Considered Harmful. In *Proceedings of the 29th Annual Hawaii International Conference on System Sciences*, pages 242–250, 1996.

[11] H. Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems. In *Proc. of the OOPSLA '86*, pages 214–223, Oct. 1986.

[12] J. D. Moffet. Specification of Management Policies and Discretionary Access Control. Chapter 17 in [15].

[13] B. A. Myers, D. Giuse, R. B. Dannenberg, B. V. Zanden, D. S. Kosbie, E. Pervin, A. Mickish, and P. Marchal. GARNET: Comprehensive Support for Graphical, Highly-Interactive User Interfaces. *IEEE Computer*, 23(11):71–85, Nov. 1990.

[14] OMG. *The Common Object Request Broker: Architecture and Specification, Revision 2.0*. Object Management Group, July 1995.

[15] M. Sloman, editor. *Network and Distributed Systems Management*. Addison-Wesley Publishing Company, 1994.

[16] M. Sloman and K. Twiddle. Domains: A Framework for Structuring Management Policy. Chapter 16 in [15].

[17] G. L. Steele Jr. *Common Lisp — The Language*. Digital Press and Prentice-Hall, second edition, 1990.

[18] Sun Microsystems Inc. NFS: Network File System Protocol Specification. Internet RFC 1097, Mar. 1989.

[19] A. Taivalsaari. *A Critical View of Inheritance and Reusability in Object-Oriented Programming*. PhD thesis, University of Jyväskylä, Finland, Nov. 1993.

[20] J. D. Thornton. Practical Descriptions of Configurations for Distributed Systems Management. In *Third Int. Conf. on Configurable Distributed Systems, Maryland*, pages 36–43, 1996.

[21] D. Ungar and R. B. Smith. Self: The Power of Simplicity. *Lisp and Symbolic Computation*, 4(3):187–206, 1991.