Towards Autonomic Networks

Alexander V. Konstantinou

Submitted in partial fulfillment of the

requirements for the degree

of Doctor of Philosophy

in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2003

Towards Autonomic Networks

ABSTRACT

Alexander V. Konstantinou

Autonomic computing has been proposed as an approach to reducing the cost and complexity of managing Information Technology (IT) infrastructure. An autonomic system is one that is self-configuring, self-optimizing, self-healing and self-protecting. Such a system requires minimal administration, mostly involving policy-level management. This thesis introduces novel results in autonomic management organization, autonomic element instrumentation, and autonomic policy maintenance. Management functions are organized in a novel two-layer peer-to-peer (P2P) architecture. The bottom layer organizes management information in a unified object-relationship model, that is instantiated in a distributed transactional object modeler repository. The top layer unifies the traditional roles of managers and elements into a single autonomic management layer. Autonomic elements use the modeler as a primary management repository, and effect autonomic behavior in terms of transactions over the shared model state. A novel language called JSpoon is introduced as a mechanism for extending element objects at design-time with management attributes and data modeling layer access primitives. JSpoon elements may be extended with additional autonomic functions at runtime using model schema plug-in extensions. This thesis further introduces a novel autonomic policy model and language in the form of acyclic spreadsheet change propagation rules, and declarative constraints. An Object Spreadsheet Language (OSL)

is introduced to express autonomic behavior as dynamic computation of element configuration over the object-relationship graph model. Static OSL analysis algorithms are presented over three incremental OSL language extensions for detecting change rule termination and performing optimal rule evaluation over any instantiation of the management model. The proposed organization has been implemented in a large prototype system that has been successfully demonstrated in security, network configuration, and active network applications.

Contents

List of Figures

List of Tables

Acknowledgments

The inspiration of this work belongs to my thesis advisor, professor Yechiam Yemini (a.k.a. YY). The development of the ideas presented also owes much to his insight. During my long circuitous path down the Ph.D. rabbit hole, Yechiam would spontaneously appear at critical junctures to provide guidance, like a Hari Seldon out of Asimov's Foundation trilogy. Yechiam's ability to drill down to the basic technical issues has been a constant source of amazement. I owe him great thanks for his trust in sponsoring my work, and his inspiration and guidance along the way.

This work also owes a debt of gratitude to the people of the DCC lab. Danilo Florissi was a great research collaborator and brought a sense of order to a group of people that were in need. Susan Tritto, a wearer of many hats, was the glue that kept our lab together, and left me blissfully unconcerned with paperwork. My fellow DCC Ph.D. students, Apostolos Dailinas, Sushil Da Silva, and Gong Su provided a challenging intellectual environment over seminars, but most commonly lunch and dinner conversations.

This work was generously funded by the Defense Advanced Research Projects Agency (DARPA) of the U.S. Department of Defense. However, financial support is only part of DARPA's assistance. Their funding brings together communities of researchers which share common interests, and creates unique research collaboration opportunities. Our program managers, Hilarie Orman and Douglan Maughan provided valuable feedback, and motivation to make our work available to our peers.

The results of this thesis were refined over several research prototypes developed as part our contractual requirements. The dreaded group demonstrations were ultimately a source of great feedback, and rich collaboration.

My thesis research, as well as, personal development as a researcher also owes much to my internships at Telcordia Technologies, and Lucent Bell Laboratories. My first note of gratitude goes to S. Rajagopalan (a.k.a Raj) of Telcordia whose trust in my research was the source of my best research collaboration. Sandeep Bhatt and Raj provided critical guidance in applying the results of my management automation research in security. Tim Griffin and Nancy Griffeth at Bell Labs afforded me the first opportunity to work in a research lab, and the opportunity to work in different research directions. Daniel Lieuwen, also at Bell Labs, provided valuable lessons in software development and documentation.

A Ph.D. thesis is not developed in a vacuum. The Computer Science department of Columbia University provided the needed physical as well as intellectual space to develop these ideas. I would like to thank professors Sal Stolfo, Gail Kaiser, Ken Ross, Jason Nieh, and Roco Servedio for their research assistance. Professor Angelos Keromytis, my second advisor, provided valuable research career advice, and the much needed impetus to graduate. I'd like to thank all the department staff for their support and friendship during these years. I would also like to thank my Ph.D. committee members, professors Al Aho, Gail Kaiser, Dan Rubinstein, and Dr. S. Rajagopalan for their valuable ideas and suggestions.

The path to writing a Ph.D. thesis in Computer Science led through RPI (Rensselaer) and Macalester College. Much of my programming languages knowledge is owed to my RPI masters thesis advisor, David Musser. Professor Michael Schneider of Macalester provided valuable career guidance at the start of this road, and by fortuitous coincidence, at the end of it as well.

The contribution of my friends to this research may not be as direct, but has

been critical nonetheless. Dimitris Bouras was my lifeline to reality. His alternating roles as fun and work personal trainer kept me on track when I was most in danger of derailment. Every summer, Dimitris would make sure that my batteries were filled for the long New York winter and the ups and downs of research. Scott Epter and his family offered me a home away from home. Scott led the way in showing the path out of the Ph.D. rabbit hole, yelling loud enough for me to follow. Ted Diament crossed my path on that Ph.D. student orientation and was an invaluable source of friendship and advice along the way. Jecky Benmayor, my adopted uncle, was there to get me started on that BASIC programming manual when I was twelve, and played a large role in my academic carreer decisions. I'd also like to thank Mark Hulber, Dimitris Itsanis, Adam Kaplan, Sam Saltiel and Stavroula Sofou for their support during the development and writing of this thesis.

My parents Vasilis and Ilana, and my sisters Ariadne and Anna have been the fuel that got me started and kept me going to the finish line. None of this work would have been possible but for their unconditional love, support, and direction. My dear grandfather Harilaos who was my progress coach along the years sadly did not live to see its completion, but never doubted the outcome. This thesis is also dedicated to my surviving grandmothers Ariadne and Rina with many thanks for their love and support.

To live is to battle with trolls
in the vaults of the heart and brain.
To write: that is to sit
in judgment over one's self
- Ibsen

To my parents.

Chapter 1

Introduction

Autonomic computing has been proposed[1] as an approach to reducing the cost and complexity of managing Information Technology (IT) infrastructure. An autonomic system is one that is self-configuring, self-optimizing, self-healing and self-protecting. Such a system requires minimal administration, mostly involving policy-level management. To effect such autonomic behavior, a system must instrument its operational behavior and external interactions with other systems. It needs to represent this information in a model which admits automated interpretation and control, incorporating knowledge on how to automate management actions. Current management architectures do not provide these essential technical features. As a result, current networks continue to depend on ad-hoc operating procedures implemented by human administrators. The high costs associated with manual management have become the most significant barrier to the scaling of technology investment.

1.1 Technical Results

This thesis introduces novel results in network management organization, network element instrumentation, and autonomic policy maintenance. Management functions are organized in a novel two-layer peer-to-peer (P2P) architecture. The bottom layer organizes management information in a unified object-relationship model, that is instantiated in a distributed transactional object repository. The top layer unifies the traditional roles of managers and elements into a single autonomic management peering layer. Autonomic elements use the repository as a primary management repository, and effect autonomic behavior in terms of transactions over the shared model state. A novel language called JSpoon is presented as a mechanism for extending element objects at design-time with management attributes and data modeling layer access primitives. JSpoon elements may be extended with additional autonomic functions at runtime using model schema plug-in extensions. This thesis further introduces a novel autonomic policy model and language in the form of acyclic spreadsheet change propagation rules, and declarative constraints. A novel Object Spreadsheet Language (OSL) is introduced to express autonomic behavior as dynamic computation of element configuration over the object-relationship graph model. Static OSL analysis algorithms are presented over three incremental OSL language extensions for detecting change rule termination and performing optimal rule evaluation over any instantiation of the management model. The proposed organization has been implemented in a large prototype system that has been successfully demonstrated in security, network configuration, and active network applications.

1.1.1 An Autonomic Management Architecture

The thesis introduces a novel two-layer peer-to-peer (P2P) management architecture for autonomic computing, depicted in figure 1.1. The bottom layer maintains a unified object-relationship model of network state and configuration. Network element status and configuration is modeled using objects, while dependencies are expressed as binary relationships between objects. Model objects may represent physical devices such as switches, routers and hosts, as well as logical services such as VLANs, IP networks, file servers, and web services. Relationships between model objects express physical connectivity, such as containment or wired linking, as well as logical connectivity, such as network-layer organization, or application-layer service dependencies. The model is instantiated in a distributed object Modeler repository which is accessed over a transactional interface. The top layer consists of network elements operating over the unified management data model. Elements use the Modeler as the primary repository for status and configuration information. All element management operations, whether internal or external, are performed over the transactional Modeler interface. The modeler exports event publish/subscribe mechanisms supporting synchronous notification within the context of the triggering transaction, as well as asynchronous out of context notification. The data model schema can be dynamically extended with semantic information. Semantic checks are performed by knowledge plug-in modules which extend the Modeler with additional functionality.

For example, consider a host whose IP network interface is configured from a pool of available addresses using the DHCP[2] protocol. Dynamic address allocation may result in different IP addresses being assigned to the same host over time. Remote access to such a host depends on the ability to resolve a persistent DNS[3] name into the current IP address. In order to maintain such a mapping, the information stored in the DHCP server, or on the host interface must be propagated

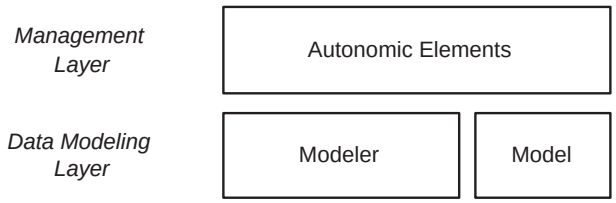

Figure 1.1: Peer-to-Peer (P2P) Autonomic Architecture Overview

to the DNS server. Presently, the configuration repositories and access protocols for IP host configuration, as well as DNS and DHCP server configurations are proprietary and non-transactional. Repository formats include flat structures stored in isolated file systems (Unix /etc files), and local platform specific hierarchical repositories (Windows Registry). Autonomic functions must be specially designed as part of these protocols, such as dynamic DNS, or through polling of configuration repositories, configuration parsing and generation over remote access protocols. In the P2P architecture, the distributed Modeler becomes the *primary* repository of configuration. The modeler stores the relevant IP interface, DHCP lease, and DNS name-address bindings in a *unified* model that is accessed over a transactional interface. Modeler events are exported over a publish/subscribe interface. In this particular example, autonomic functions will be triggered by a change in host's the IP interface configuration and can be propagated to the corresponding name-address DNS configuration by navigating the model's service relationships. The autonomic functions occur within the context of the triggering transaction, presenting in consistent configuration views.

The P2P management architecture supports *concurrent* multi-manager control of network elements. Ownership of management attributes is removed from network elements and assigned to a distributed database, thereby enabling the application of traditional database concurrency control techniques. The unification

of manager-element roles improves *safety* by eliminating the state synchronization problem between managers and elements. The elimination of management agents through direct element instrumentation improves *reliability* through reductions in the size and complexity of implementing managed network services. Transactional management attribute logging and locking furthermore create *recoverable* configuration semantics.

The P2P management architecture also provides *scalable* monitoring and control of network elements. Management functions can be safely distributed across multiple managers due to the protection of transactional concurrency control. The unification of the manager and element roles in a peering relation enables the delegation of management functions, effectively distributing management load, and supports *self-healing* in the face of local network failures. The unified management model can be extended by managers with additional information, such as link-layer topology. Transactional model access assures that elements and other managers can utilize this additional information to effect *self-management* and *self-organization*.

Another important contribution of the P2P architecture is an extensible mechanism for associating *semantic information* with the management model. The unified management model admits a restricted set of operations over a transactional context. Any management change can thus be intercepted synchronously by another manager in order to verify policy, or propagate change. Such semantic verification and propagation policies can be attached to the unified data model, and automatically enforced by the repository through a plug-in mechanism. Automated policy enforcement creates a *reliable*, *efficient*, and *secure* autonomic management environment.

1.1.2 A Language for Autonomy

This thesis introduces a novel approach to element management instrumentation in the form of design-time language extensions. Autonomic elements are developed in a new language called JSpoon which extends the Java language with autonomic management features supporting management attribute declarations, access synchronization primitives, and event subscriptions. The JSpoon runtime further supports management persistence, and dynamic schema semantic extensions in the form of knowledge plug-ins.

Figure 1.2 illustrates key JSpoon features in the context of an autonomic network time service example. Management attributes are identified using special JSpoon declarators. The JSpoon object is compiled into a Java class with the non-management Java attributes and methods, as well as accessor methods to the JSpoon configuration attributes. An additional management class is introduced which exports the management attributes through accessor methods of other JSpoon programs. At runtime, programs accessing configuration attributes interact with the JSpoon runtime environment which provides persistence, concurrency control, remote access, and event notification services. The management schema of JSpoon services may be extended dynamically with semantic information. For example, a constraint may be associated with the port number, as shown in the figure. The JSpoon runtime will invoke the appropriate plug-in to evaluate the expression after the value has been changed.

JSpoon simplifies *service development* by automating the export of management attributes. Developers are thus encouraged to expose all configuration attributes resulting in an overall improvement in *service manageability*. Automation further reduces the need for programmers to deal with multiple copies of a configuration attributes, for internal, persistence and standard MIB support, thereby improving *safety* and *consistency*. Transactional access to configuration properties

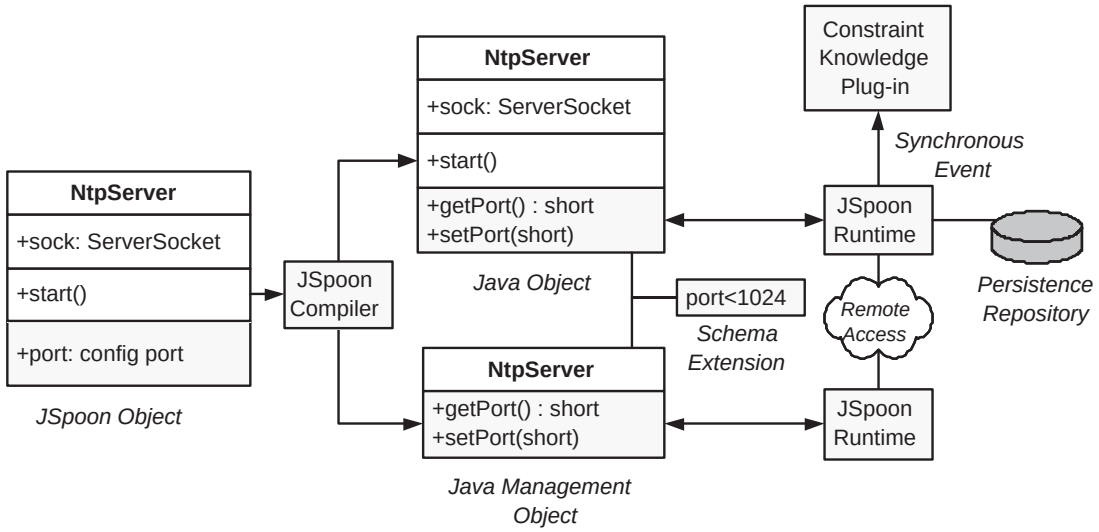

Figure 1.2: JSpoon Feature Overview

creates a safe environment for *concurrent management* of network services. JSpoon relationship declarations enable the linking of related services creating a mechanism for service *discovery*.

JSpoon management events extend Java with a new *dynamic service extension* capability. Synchronous events are a mechanism for extending the behavior of services to provide additional change propagation, and configuration verification capabilities. This capability *generalizes exception generation and handling* from a single thread of execution to a set of distributed processes. Asynchronous management events provide improved change *event correlation* based on the logging of changes propagating across multiple elements.

JSpoon schema plug-ins support runtime *data and semantic extensibility*. Access to management information is necessary in a variety of applications, such as configuration automation, performance and fault monitoring, and inventory control. Information collected from JSpoon elements can be used to dynamically *enhance the configuration model*. Semantic schema extensions can provide *protection* from inop-

erable, or inefficient configurations based on the enhanced configuration model. Service behavior may be *customized* to local operational requirements through change propagation plug-ins. This approach creates *new markets* for systems services for plug-in vendors, and plug-in service providers.

1.1.3 Effecting Change Propagation

This thesis introduces an Object Spreadsheet Language (OSL) to express change propagation over object-relationship configuration models. OSL change rules are expressed as *spreadsheet-style* [4] computations over the configuration model, as shown in figure 1.3. In the figured example, the status configuration property of a hosted application is determined by the configuration state of its host. A change in the host status configuration will thus propagate over the `serves` relationship to the application object. Autonomic element behavior results from the application of composable OSL libraries. The thesis presents algorithms for static, compile-time analysis of change rules to assure that autonomic elements maintain configuration safety within the dynamic environments in which they are installed. Propagation analysis must be performed at design time to assure safety. The results of static analysis are used by algorithms for efficient computation of change rules. A declarative subset of OSL called the Object Policy Language (OPL) is used to express and enforce configuration policy constraints. Policy constraints are a tool for protecting systems from failures due to invalid propagation of changes. The thesis introduces mechanisms to support scalable, multi-domain autonomic configuration management and control of the scope of propagated changes. These mechanisms will be used to coordinate configuration changes among independent domains.

The spreadsheet approach to change propagation provides a *simpler* mechanism for programming autonomic functions than a general purpose programming languages. The spreadsheet model has already proven successful in allowing non-

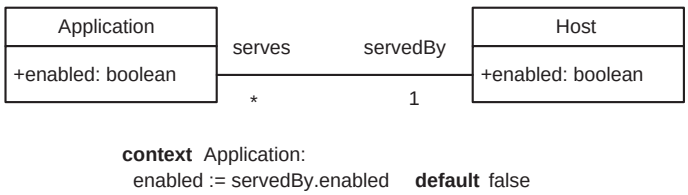

Figure 1.3: Object-Relationship Spreadsheet Model

programmers to express financial computation as acyclic propagation over grid data. Spreadsheets define the value of variables as a function over other variables. Users do not need to reason about previous variable state, since it cannot be introduced as part of the right-hand-side expression. Spreadsheet rules can thus be viewed as declarative constraints over a the value of a set of variables. The spreadsheet approach has been demonstrated in a variety of configuration automation in security, service self-configuration, and mobility applications.

The OSL language was designed to support expression of management change propagation over an object-relationship model, while remaining *statically verifiable*. Changes in the spreadsheet model must propagate without cycles in rule evaluation. Cycles may lead to infinite computation, and invalidate the benefits of simplicity due to the lack of looping constructs. Unlike financial spreadsheets, which typically detect cycles at runtime, autonomic applications cannot depend on user intervention, therefore static termination verification is essential. All OSL rule sets can be analyzed statically to determine if there exists a model instantiation which will lead to cyclical evaluation. The static analysis depends on construction of a triggering graph which can be analyzed in linear time.

The static analysis of OSL rules may be used to effect efficient *incremental rule evaluation*. The spreadsheet model semantics require that any change in variable value result in recomputation of all functions dependent on the changed variable. OSL expressions can be analyzed to determine the optimal rule evaluation

order, as well as the minimal set of object instances over which the rules need to be evaluated. The algorithm depends on the results of the static rule analysis phase.

The declarative Object Policy Language (OPL) can *prevent policy violations* due to change propagation *before* they take effect. Change propagation is necessary to support autonomic self-management, self-healing, and self-optimization functions. However, certain reactions to changes may lead to operational state, or improved operation by violating policy. For example, a security sensitive service may react to the failure of its authentication server by enabling unauthenticated access. OPL expressions can detect such policy violations, and abort the transactions before their effects have affected element behavior.

The domain mechanisms introduced establish *bounds* on change propagation, as well as *scalable and safe* change propagation across domain peering points. Propagation across domains is performed over views of shared elements. Propagation paths over shared view objects are summarized in order to support scalable rule analysis and maintenance. The peering view defines the types of permitted propagation, and can be the subject of policy rules. Policy rules increase safety by further filtering the semantic content of propagated values.

1.2 Rationale for Autonomic Systems

1.2.1 Practical Challenges

Current networks are operated using ad-hoc procedures implemented by expert human administrators. Change management in such an environment is human intensive, slow, and can result in unpredictable failures and inefficiencies requiring costly recovery. The overall cost of current management practices is estimated to occupy over 70% of corporate Information Technology (IT) budgets[5]. Current networks cannot be enlarged without a corresponding quadruple increase in man-

agement costs. As a result management has become the most significant barrier to scaling technology investment.

The major components of management overheads are human resources costs, down-time costs, opportunity costs due to slow service deployment, and user-training costs. Management automation can significantly impact each of these components. Automated networks may be expanded without equivalent growth in payroll costs. Down-time can be reduced through automated policy enforcement, and systematic recovery. Service deployment and element configuration can be performed faster outside direct human control. Finally, user training can be significantly reduced since a large part of it involves systems management, and failure handling.

Management Example

The practical challenges of current network practices will be illustrated by a change management example of installing a new web-based application. Web services [6], are currently structured in multiple tiers, as depicted in figure 1.4. Web clients transmit application requests over the HTTP protocol. Requests are received by a proxy service which redirects traffic to the appropriate HTTP server based on the URL target, and current work load measurements. The web server assembles the presentation of its response by combining static and dynamically generated content. Dynamic web-server content is generated by servlet procedures which perform calculations over data retrieved from a business logic layer. The business logic layer provides structure and control over the raw enterprise data in manner that preserves business policy. Finally, the enterprise data layer consists of back-end relational database and legacy applications.

Consider the task of installing a new web-based application in this multi-tiered architecture. The application servlet program must be installed on the web-

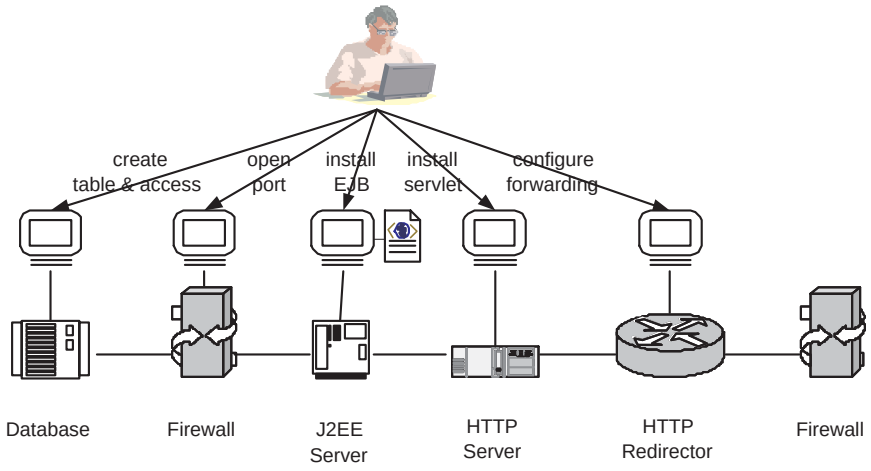

Figure 1.4: Web-Based Application Configuration Example

server host, and the web-server must be configured to map it to a specific URL location. In installations, consisting of multiple web-server hosts, the application might only be installed in a subset of the available hosts, requiring reconfiguration of the web-redirector. The application may further require additional business logic services, or upgrades of existing services. These dependencies may conflict with dependencies of previously deployed applications, which may restrict the hosts in which the updated business logic services can be deployed. New business logic services may require access to previously inaccessible databases, requiring reconfiguration of the internal DMZ firewall. Finally, the new application may require establishment of database authentication credentials, as well as database table creations.

Today, all these changes must be performed by human administrators who are responsible for evaluating the required changes in terms of operational policy, establishing a change work-flow plan, and effecting change by independently updating the low level configuration repositories of each effected element. These tasks may require coordination with different managers or management groups that

have been assigned ownership of different element domains. For example, installing the web application will require identifying the target web-servers where it can be safely installed, as well as the target hosts of the required business logic services. The web-server XML configuration file will need to be locked, inspected, modified, verified, and reloaded. Firewalls are typically managed by a separate group, and opening the port may require negotiations over application requirements and existing security policies. The order of changes may be restricted based on security and availability requirements. Removing a fully or partially installed application is an equally involved process, with the added risk of disturbing undocumented dependencies, or leaving unnecessary configuration state. Considering that large sites may deploy thousands of such applications, it is clear that the complexity and cost of human management present a major barrier to system scalability.

1.2.2 Technical Challenges

Attempts at automating network operations have so far met with limited success due to the design of existing management architectures. Current architectures assume a manager-agent (client-server) model in which element performance and status information is presented to human managers. Managers must collect and interpret this information in relation to network policy. Policy enforcement requires manual change management over distributed, heterogeneous element configuration repositories. Managers are further required to manually log and coordinate configuration updates across multiple elements due to lack of transactional configuration access mechanisms. These architectural limitations create significant safety, scalability, and reliability challenges to automation.

Several factors make the design of self-configuring networks under the current management architecture challenging:

1. The *change propagation problem*: A configuration management task typically

requires changes in multiple interdependent elements. Deploying the web-based application, in the previous example, required installing business logic adapters, opening ports in the back-end firewall, and reconfiguring the web redirector. Self-configuring software needs to:

- Recognize these different elements, their relationships and configuration states – *network topology discovery*. Currently, this information is spread in fragmented repositories and can be very difficult to compile.
- Represent the knowledge of the sequence of changes in these elements – *change propagation rules*. Automation logic is currently expressed in general-purpose scripting languages which cannot be statically analyzed to determine their effect, before the change is applied.
- Effect the changes in each element through heterogeneous widely varying proprietary instrumentation, configuration tools and operational procedures; and coordinate these changes with those caused by built-in element procedures – *handle element heterogeneity and spontaneous changes*; and
- Enable recovery and undoing of changes, in case of failures – *recoverability*. Current management protocols do not support transactional mechanisms, and therefore an external agent cannot enforce isolation, or automatic recovery.

2. The *configuration policy problem*: Configuration changes may lead to inconsistent configuration states resulting in operational failures and inefficiencies. Therefore, a self-configuring network needs to:

   - *Represent* policy knowledge about configuration consistency relationships in the form of policy constraints. Current management systems[7,

8] do not define unified models. Management Information Bases form "data islands" which are not bridged in the model. Policy languages depend on relationships expressed as part of the static schema, and cannot be used in such an environment.

- *Enforce* these policy constraints to assure consistent configurations. Policy enforcement requires synchronous control of change propagation. Changes detected through monitoring have already been committed, may not be recoverable, and their transient effect can result in permanent security compromises.
- Enable organizations to *program* policy constraints to effect their operational policies.

3. The *composition problem*: A self-configuring network needs to adapt the change propagation rules and policy constraints to the network configuration. It must compose these rules and constraints from component change propagation rules and policy constraints associated with individual elements. For example, the installation of an upgraded business logic component may require the upgrade of dependent web applications. Script-based change propagation and verification tools are not composable, since it is not possible to analyze their dependencies or combined effect.

1.3 Thesis Organization

The thesis is organized in four major chapters. The second chapter introduces the proposed two-layer peer-to-peer (P2P) management architecture and presents its technical advantages over existing three-layer Manager-Agent (MA) architectures[7, 8]. The third chapter presents the JSpoon language for embedding P2P data modeling layer access into autonomic elements, and presents the technical advantages

of the language-based approach to element instrumentation. The fourth chapter presents the OSL language for spreadsheet-style change propagation and policy declaration over the object-relationship data model. The chapter presents algorithms for static analysis of OSL rule sets, and efficient rule evaluation, and details the domain-based mechanisms for scaling rule and police evaluation. The fifth chapter presents the application of the NESTOR P2P architecture prototype to automation of security, and active network management. Related work discussion is presented at the conclusion of each chapter. The thesis ends with a brief conclusion chapter.

Chapter 2

An Architecture for Autonomy

2.1 Introduction

This chapter introduces a novel peer-to-peer network management automation architecture. In the peer-to-peer architecture, elements and managers operate over a unified management model. Elements access the configuration model to retrieve their current configuration and export their state, while managers use the model to discover element topology in order to effect change. Configuration models are expressed in terms of classes of objects which are linked by relationships. The management model is instantiated in a distributed object repository (Modeler) which is accessed through a transactional interface.

The proposed peer-to-peer organization offers significant advantages over the traditional manager-agent (client-server) organization. The unified model allows managers to discover, access and manipulate the configuration of all network elements. Transactional access to management information creates an environment supporting safe multi-manager access, as well as recoverable configuration change semantics. The unification of the traditional roles of manager and element allows management functions to be distributed in different elements, supporting

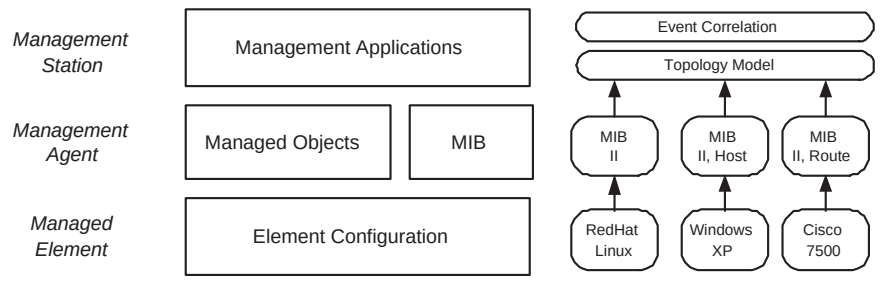

Figure 2.1: Manager-Agent Management Architecture

autonomic behavior. Transactions establish natural policy enforcement points, and can be used to more accurately correlate the root cause of network failures or inefficiencies. Distribution of element configuration creates a scalable management infrastructure, which can continue to operate under network partition, to maintain policy, and effect self-healing.

2.1.1 Management Architecture Automation Challenges

Current network management standards [7, 8] are organized in a three layer manager-agent (client-server) architecture depicted on the left side of figure 2.1. At the top layer, centralized management applications access the configuration of distributed network elements through a set of standardized schemata called Management Information Bases (MIBs). MIBs encapsulate the local configuration, performance, and status of individual network elements. Management applications must correlate the information collected across multiple elements to determine network behavior, and effect network control. Each network element is associated with a management agent which collects management information using proprietary interfaces, binds it to one or more management MIBs, and communicates with the management application via a standardized network protocol.

The right side of figure 2.1, depicts an example of a manager-agent manage-

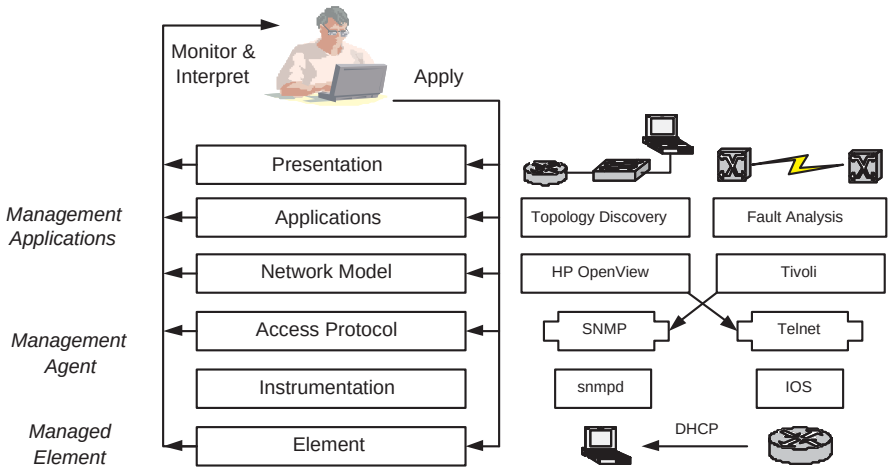

Figure 2.2: Manager-Agent Management Platforms

ment fault root cause analysis automation application. Networks generate many types of related events in response to changes in configuration, or usage patterns. The root cause is determined by correlating network events with network topology and signatures of known problems. In the manager- agent architecture, the first step involves discovery and collection of MIB information from network elements. Data collection requires the establishment of authentication credentials with each element. Discovered elements must be polled for change events, and the network must be polled for new elements. In the next step, the collected information must be mapped into a topology model to express dependencies between elements at multiple layers. Finally, the topology model and detected events can be provided to the correlation engine for analysis.

Discovery is a common requirement for many other management applications, such as inventory control, topology visualization, and performance optimization. Management platforms have evolved to address this common application requirement, at the cost of establishing proprietary element configuration models and repositories. The architecture of current management systems is depicted in

figure 2.2. At the top layer, managers monitor presentation software for failure events. Presentation software is usually tightly bound to a specific application, such as topology discovery, or fault analysis. Applications contribute information to a common repository which is management vendor-platform dependent. The repository is instrumented by adapters which communicate to the element agents over a network protocol and map the information retrieved into the network model repository.

Current discovery solutions cannot address all the needs of monitoring management applications. Element discovery repositories are proprietary and do not support all element types, resulting in information fragmentation. Threads of change in element configuration cannot be correlated because of the non-transactional nature of the protocols used to populate the network model. As a result, management applications must apply additional task-specific correlation functions in order to improve presentation consistency.

Furthermore, these discovery solutions offer a weak foundation for configuration management applications. Due to the lack of atomicity and isolation functions at the agent-level, it is not possible to obtain a consistent view of configuration in a fast changing system. Even when consistent views can be obtained, they cannot be locked since other management platforms, human managers, or dynamic configuration protocols can effect change at any point. This is an instance of consensus problem [9] which cannot be solved in asynchronous systems [10]. Therefore, current management cannot guarantee policy enforcement. Policy violations due to race conditions can result in transient failures effecting performance, or hard failures requiring manual intervention. Transient failures in security create can be exploited to effect permanent system compromise.

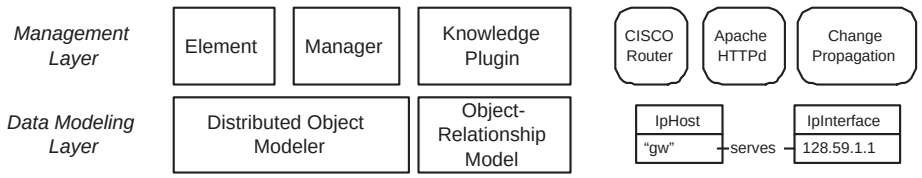

Figure 2.3: Peer-to-Peer Architecture Example

2.2 A Peer-to-Peer Management Architecture

In order to support autonomic behavior, a management architecture must satisfy certain basic requirements: (1) support the representation of element configuration and performance properties used to control and monitor element behavior, (2) express relationships between different autonomic elements, (3) control access to configuration properties so as to assure consistent views, (4) enable autonomic elements to discover, access and control the configuration of other dependent elements, (5) provide publish-subscribe interfaces for management event notification, and (6) enable element configuration persistence and recovery.

The proposed peer-to-peer (P2P) management architecture was designed to address these requirements. The P2P architecture organizes autonomic systems into a two-layer architecture as depicted in Figure 2.3. At the bottom layer, a distributed object *Modeler* provides a consolidated element data repository, including configuration, relationship, state and performance attributes as well as their behavior events. Modeler objects are instances of classes declared in a unified management `model`. The Modeler provides interfaces to access and manipulate the managed data. This enables the management layer, above, to access a unified data model, interpret its behavior and activate autonomic control functions.

2.2.1 Modeling Management Information

Element configuration and performance has been represented in a variety of models. Flat structures were the earliest management models and had the form of attribute-value pairs. Flat models cannot representing structured and tabular information, such as a route table, which is a common modeling requirement. Hierarchical tree models were thus adopted by the SNMP[8] standard. Hierarchical models offer limited extensibility, in the form of tree branching. Extensibility, in the form of function specialization is a common management modeling requirement. Object-orientation combines structured data representation with an extension mechanism called inheritance. Therefore, most subsequent management standards have been based on object-based management models [11, 12, 13, 14, 15, 16].

Relationships are an object-oriented modeling mechanism to establish associations between objects. Relationships define binary associations between classes of pre-defined cardinality. For example, a `Host` single object may be associated with one or more `IpInterface` objects. While relationships are commonly used in modeling environments, such as UML[17], they are not included in the type systems of popular object-oriented languages, such as C++ and Java. Although it is possible to model them with pairs of pointers, or sets of pointers, this approach places a burden on programmers and can result in inconsistent relationship endpoint memberships.

Configuration models in the P2P architecture are expressed in the Resource Definition Language (RDL). RDL is an object-oriented interface language that supports the specification of resources as objects and their relationships. Object-orientation provides important clustering of configuration and behavior through interface inheritance and hierarchy mechanisms. Interfaces define generic behaviors of objects and inheritance supports abstraction of common features. Relationships between objects capture interdependencies through hierarchical structures, as

```
interface NetworkHost {
  attribute String hostname "Name of host";
  relationshipset interfacedThrough, NetworkInterface, partOf;
}
interface NetworkInterface : netmate::Node {
  key attribute byte[] uniqueIdentifier "e.g. MAC Address";
  relationship partOf, NetworkHost, interfacedThrough;
}
```

Table 2.1: Resource Definition Language (RDL) Model Examples

well as of distribution. Finally, objects encapsulate the methods for accessing the underlying element instrumentation.

Autonomic management instrumentation variables can be assigned to one of three basic categories[11][18], with associated access patterns. *Configuration properties* control the behavior of the autonomic element and must therefore be protected in regards to concurrency and semantic content. *Performance properties* export element performance measurements and operational state, cannot be locked, and may only be set by the element owning the object. *Relationships* express dependencies to other autonomic elements. State information is divided between configuration properties which express intention, and read-only performance properties which express operational status. Operational state can only be indirectly effected through configuration state changes.

Table 2.1 depicts fragments depicts fragments of the model of an IP host expressed in RDL. Interfaces are pure abstract classes, which may be scoped in a package. Packages are a requirement in an environment where models are likely to be imported from external sources, such as vendors or standard bodies. Interface definitions may include attribute, method, and relationship declarations. In the `NetworkHost` example, the first statement declares a string attribute named *hostname*, which represents the name of the modeled host. The second statement declares an any-to-many association between this interface and classes im-

plementing the interface `NetworkInterface`. Associations are declared by naming both ends (role names), the type of the association class, and the multiplicity of the association (one, or many). In the example, the association between `IpHost` and `NetworkInterface` is specified as one-to-many. The model reflects the fact that objects of type IP host may have one or more IP interfaces. The relationship `partOf` goes in the other direction, from an `NetworkInterface` to a `NetworkHost`. The `netmate::` scope in the declaration of `NetworkInterface` denotes the NETMATE[11] schema which is used as a basis for RDL models.

These resource models constructed using RDL incorporate essential information for self-management and self-organization that is otherwise hidden in obscure operational manuals, requires complex discovery mechanisms, or is just unavailable. The models enable simple, uniform, and secure access and manipulation of resource information. For example, consider the `hostname` attribute of the `NetworkHost` interface. The method for accessing and updating the name of a host is platform-dependent. Moreover, it may involve multiple operations, such as updating a configuration file and then invoking a system utility to update the operating system data structures. In some cases, the modeled element may not even support a name attribute, and the value may be stored in third-party repository. By viewing configuration through the unified model, all this complexity can be hidden, enabling managers to focus on the task at hand.

2.2.2 Instantiating the Management Model

In the P2P architecture, autonomic elements instrument their configuration and performance management information by instantiating the object-relationship model classes in an object Modeler. The Modeler is an object repository distributed among the network elements. Elements typically maintain a local object repository to assure access to configuration in the absence or failure of network connectivity. The

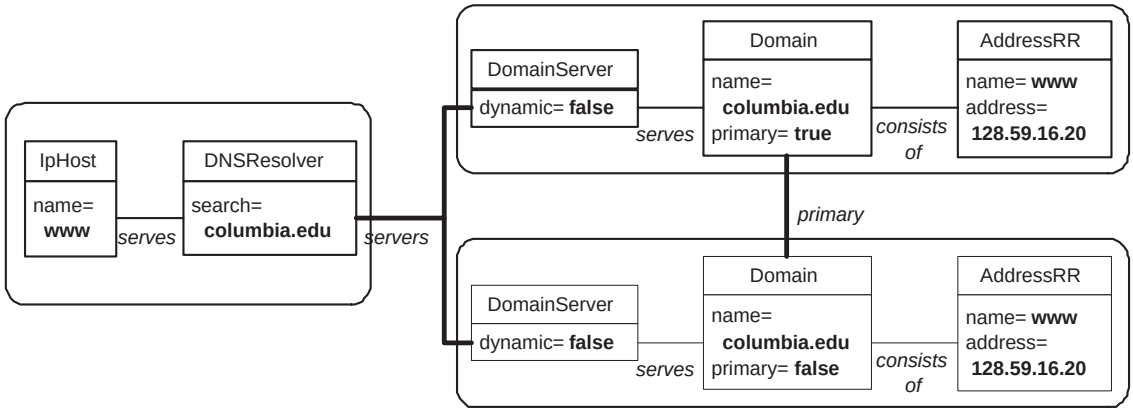

Figure 2.4: Distributed Object Modeler

distributed repositories are linked by relationships that connect objects residing in different repositories. Every object is a associated with an authoritative repository. Objects may be transparently replicated to improve read access performance, or to provide management views.

Figure 2.4 shows a sample instantiation of a management model (not shown) that is distributed in three repositories. The repository on the left side stores the management objects of a simple Internet host. As shown, the host has a name, and is served by a a DNS[3] resolver which performs domain-name to IP address translation. The resolver is related with one or more domain name servers which provide the service. In this example, the resolver is related to two domain server objects residing in different repositories (relationships connecting objects in different repositories are emphasized). Each domain server is related to domains which it serves as primary, or secondary, and each domain contains resource records with the appropriate mappings. A secondary domain is related to its primary to model the fact that changes to the primary propagate to the secondary.

Operation	*Operands*	*Description*
Load	class	Adds a new class to the model.
Unload	class	Removes a class from the model. The repository must not contain any class instances.
Create	object	Creates a new instance of a model class. All repository objects are associated with a lease.
Remove	object	Removes a class instance. All the instance relationships must be empty.
Set	object, property	sets the value of an attribute or relationship. To-many relationships are set by assigning a set or sequence of elements.
Retrieve	object	Retrieves an object based on its unique identifier.
Lookup	class	retrieves all instances of a class.
Get	object, property	Retrieves the value of an object attribute or relationship.

Table 2.2: Modeler Operations

Controlling Access

Modeler repositories are used to store and access management information in the form of object class instances, and relationships establishing binary associations between objects. The Modeler admits a limited number of primitive object operations that are listed in table 2.2. Higher-level operations may be built on the lower-level ones such as searches based on attribute value or relationship membership.

The Modeler operations can be mapped into relational or object distributed database operations. Due to this mapping, database concurrency control and recoverability techniques can be applied to the Modeler[19]. Concurrency control can be maintained using the three-phase commit distributed transaction protocol. Transaction atomicity and isolation can be maintained with two-phase locking. Transaction log ordering can be performed by maintaining logical clocks in each transaction participant.

Discovering Management Information

An autonomic service must be capable of discovering its environment in order to perform its self-management and self-healing functions. The P2P architecture supports discovery through relationship navigation. Autonomic elements may navigate external relationships to their local model in order to discover related services. For example, an autonomic DNS resolver would navigate the local host model to identify the network interface objects. Each local network interface is related to its peers which can be accessed to query services running on the remote hosts. The remote host objects would then examined for DNS services, and ranked based on performance-based characteristics.

Relationship-based discovery depends on the establishment of some initial relationship that crosses repository boundaries. Such initial relationships may be hard-coded, in the form of element identifiers. Alternatively, autonomic element discovery protocols, such as DHCP[2], Jini[20], UPNP[21] can be used to establish network membership. Network membership can be used to discover available services.

Monitoring Management Information

Each primitive database update operation identifies a primitive change event: *load*, *unload*, *create*, *remove*, *set*. In addition, the transaction mechanism creates an additional primitive events: transaction *commit*. Managers can subscribe with the repository for primitive events. Repositories also support subscriptions for event filtering based on key and non-key attribute values and relationship membership.

Modeler event notifications may be received synchronously or asynchronously. Synchronous notification occurs in the context of the event triggering transaction. The event triggering thread is suspended until each event subscriber has been notified. Recipients may perform additional Modeler operations in the context of

the transaction, or may request a transaction rollback (abort), in their notification handler. Asynchronous event notifications are delivered outside the triggering transaction context. Recipients may examine the event transaction log to determine the context of the event.

Subscribers receive asynchronous event notifications in an order that preserves the transaction serializability. Events triggered from the same transaction are delivered in the transaction log order. Events triggered by different transaction are delivered in serial history order. Results from messaging systems research on message delivery and filtering can be applied for this task [22, 23].

Persistent Configuration

Configuration typically needs to persist across service activations. Currently, configuration persistence is separately handled by each element. The P2P architecture supports both element as well as Modeler supported persistence. Elements may independently persist their information. Element objects can be explicitly removed from the repository when the service is shutdown, or may timeout when their leases are not renewed. Alternatively, elements may utilize the Modeler persistence mechanisms. Modeler persistence requires that at least one service object be associated with one or more key attributes that can be uniquely determined at service startup. For example, a web-service can retrieve its configuration by looking up the service port number. Modeler persistence supports off-line element configuration. Off-line configuration assures that the service will start in a state that is consistent with changes that occurred while it was unavailable.

2.2.3 Integrating Services

The P2P autonomic management architecture requires significant redesign of existing network elements. The next two chapters will introduce results aimed at

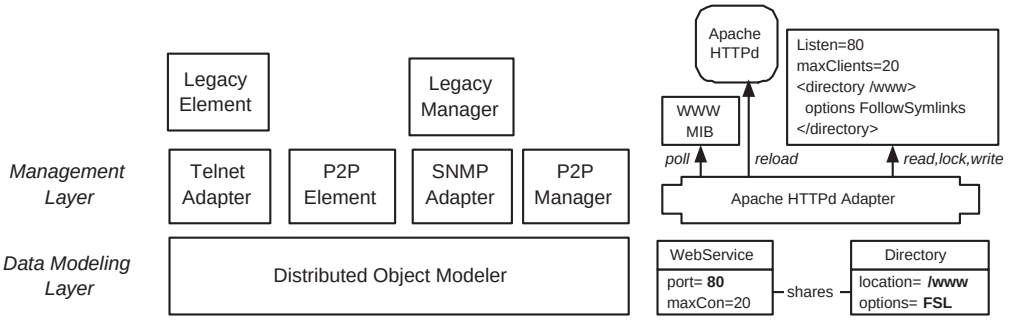

Figure 2.5: Resource Adapters

simplifying this task in the form of a language for embedding autonomic element instrumentation, and languages for expressing policy constraints, and change propagation rules. Existing Manager-Agent (MA) architecture services may partially benefit from the P2P architecture through the use of element adapters, as illustrated in figure 2.5. Adapters can provide bidirectional mappings between the MA configuration and performance models and the unified model stored in the object repository. Adapters can be used to instrument elements as well as managers.

An element adapter instruments the Modeler with configuration extracted from an MA service through standard management protocols, as well as custom proprietary configuration adapters. For example, an adapter for the Apache HTTP service, as shown in figure 2.5 will parse the `httpd.conf` configuration file, map the information into the management model object-relational schema, and create the appropriate objects in the Modeler. Performance information is gathered using the SNMP WWW-MIB[24]. The adapter also establishes a synchronous subscription on the Modeler objects it has created. Changes to these objects must be propagated to the native configuration repository, and the service must be asked to reload its configuration.

The challenges of adapting elements and managers to a unified repository have already been covered in this chapter's introduction. Obtaining exclusive ac-

cess to the adapted element configuration is not usually possible, since element configuration protocols do not support locking, and elements may support multiple configuration mechanisms (file, web, and others). Changes that occur due to element self-management functions, such as DHCP configurations, OSPF/RIP route updates, or UPNP discoveries, cannot be assigned to a specific threads of change. Reloading configuration can disrupt normal service behavior, resulting in transient failures. Performance instrumentation is limited to MIB variables, and updates limited to the rate of polling. Changes may go undetected by polling due to their transient nature, or rapid rate of change.

2.3 Coordinating Management Operations

The P2P management architecture supports a greater degree of management coordination. Control of management access and storage is transfered from the element into a distributed object Modeler repository. Elements and managers use the same transactional interface to export and access the stored management data. This approach enables concurrent access to management information while maintaining safety, and reliability. This section will present the P2P Modeler concurrency control mechanisms, and illustrate the ways in which they improve safety and reliability.

2.3.1 Concurrent Access

Traditional management architectures do not provide mechanisms for atomic, consistent, and isolated access to element management information, or manager-to-manager coordination. For example, in order to provide safe manual configuration of IP addresses in a subnet, a single manager must monitor and control the IP interface configuration of all subnet hosts. Therefore, in practice, control and mon-

itoring of network elements must be restricted to a single domain-wide centralized manager.

The P2P management architecture supports *concurrent* multi-manager control of network elements. All management information is placed in a distributed object-relationship database supporting transactional access. This approach addresses the concurrency control issue at a low level, alleviating the need for explicit manager-to-manager communication. Managers may safely operate on the same configuration sources through database mechanisms that maintain atomicity, consistency, and isolation. Element self-management functions which can be a source of risks in MA architectures, are similarly protected.

Element management differs from traditional database applications in two significant aspects. Management performance information changes at a rapid rate, and cannot be rolled-back in the sense that it monitors operations, rather than affect change. In contrast, configuration changes occur at a slower pace because they control the process, and so effect future operations. As a result, performance management attributes are not processed in a transactional manner. The second difference concerns the modeling of failed services. A failed service cannot be reconfigured until it has been restored. Examples include software systems that have been rendered inaccessible through erroneous configuration, or physical services that have been physically reconfigured, damaged, or destroyed.

Partitioned operation and recovery have been studied in the context of distributed database systems [25, 26, 27, 28]. The emphasis of that work has been on replicating data while maintaining a serializable history when the partitions are merged. In contrast, replication of network services is performed at the logical level, rather than the data representation level. Physical services cannot be replicated by software, while software services can be replicated through relocation, but their configuration will not be identical. Therefore, the configuration of an individual

network element cannot be replicated for the purpose of write updates.

The P2P management Modeler failure recovery mechanisms will be illustrated using the earlier example of an autonomic DNS resolver. The configuration of such a resolver is computed based on the host IP layer connectivity to local subnet DNS servers. To maintain configuration consistency, the resolver subscribes for changes in network connectivity, and server status. Consider a failure in a physical link of the host, or the switch to which it is connected. Through polling as well as ICMP monitoring the resolver repository will detect that one or more remote repositories are no longer reachable. These repositories were monitored because their objects were related to local repository objects.

In a distributed database, the state of these unreachable objects would be either frozen, or optimistically replicated until connectivity could be restored. Neither locking nor optimistic replication are appropriate solutions in management automation. Locking precludes corrective action on the exact services that are most likely to be dependent on the connectivity that is lost. Replication for the purpose of maintaining relationships can generate inconsistencies between the model and the real world, resulting in additional failures. Conversely, strict serializability-maintaining replication for the purpose of removing the relationships is most likely to require extensive rollback when connectivity is restored.

In the P2P management Modeler, remote objects whose repositories are no longer accessible are removed from every local object relationship. First, any active transactions which have obtained locks on the affected relationships must be aborted or committed based on their current state. The freed local relationship locks are then assigned to a recovery transaction that removes all unreachable remote objects from the local relationships. The effects of the recovery transaction may trigger synchronous event handlers. These handlers may attempt to effect self-healing by reconfiguring the network, or may attempt to abort the transaction

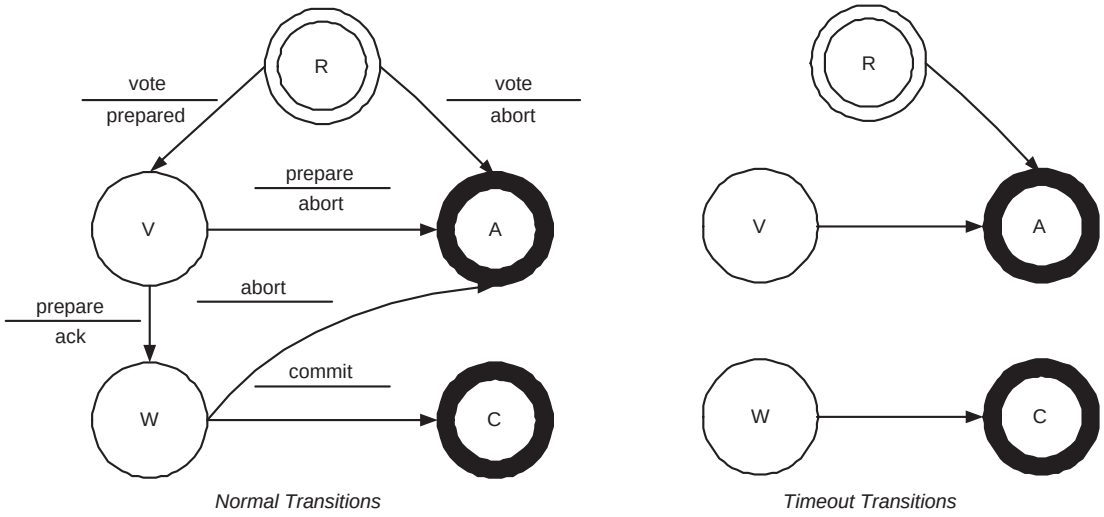

Figure 2.6: Three-Phase Commit (3PC) Participant State Diagram

to maintain policy. If the recovery transaction is aborted by a synchronous handler, then the Modeler is said to be in an *inconsistent state*. Inconsistent systems must rely on an external intervention in the form of human management, or autonomic failure recovery by the other element repositories.

The three-phase commit protocol participant state transition diagram is shown on the left side of figure 2.6. Transactions in the active state R can be safely aborted, with the transaction manager receiving asynchronous abort notification. Transactions in the voting state V can also be safely aborted with asynchronous coordinator notification, since the *prepared* vote has to be acknowledged. In the absence of failures, the transaction manager will always commit transactions for which all participants have voted *prepared*. Therefore, transactions in the waiting state W cannot be later aborted due to a policy violation. Recovery from state W involves discovery of the partitioned topology, and application of the change propagation rules to effect local consistency.

2.3.2 Safe Access

Traditional MA architectures assign ownership of configuration information to the individual elements. Network elements typically embed their management information as part of their internal programming code and structures. Because elements are coded in languages without native transactional support, MA elements are limited in their ability to synchronize and rollback management changes. Moreover, the prevalent method of MA element instrumentation creates an additional management modeling layer which creates data duplication and further synchronization risks. Therefore, in practice, current network elements cannot *safely* support concurrent management access.

Another MA architectural assumption with effects to safety is that element configuration is controlled by centralized management applications. The emergence of dynamic configuration protocols [29, 2] has invalidated that assumption, since elements may modify their own configuration in response to changes in network topology, or lease expiration. This two-way propagation of change creates race conditions in the decision processes of managers and self-configuring agents, which are exacerbated by polling mechanisms. Moreover, network elements typically support additional mechanisms for configuration, such as configuration files, which may be used to circumvent the protections employed by the centralized manager.

The P2P architecture creates a *safe* management operational environment by restructuring the manager-agent state synchronization problem as a database operation. Management information is removed from direct element control and placed in a transactional Modeler repository. Elements and managers access management data using the same narrow transactional interface. Database synchronization techniques guarantee atomic commitment of changes, consistent views, and manager isolation. In this manner, the P2P architecture supports safe multi-manager access, which further enables safe distribution of management functions

into network elements. The later capability is essential in establishing autonomic element behavior.

2.3.3 Reliable Access

The MA management architecture does not place strict controls on the types of protocols used for monitoring and configuration. A typical network element will support multiple mechanisms for accessing its management state. For example, a CISCO router may be monitored and configured using SNMP, Telnet console access, as well as a Web-based management interface. Managers may forced to use multiple protocols, some proprietary, to extract the required information, and effect configuration change. The burden of supporting additional interfaces increases the complexity of coding and maintaining agent and element implementations. Element evolution presents the possibility of mismatched semantics between manager and agent potentially resulting in the export of erroneous information, and the failure to effect durable configuration changes. These problems are exacerbated when management agents are provided by third parties.

For example, consider the management configuration task of persistently changing the name of a Linux host. The Linux runtime stores the hostname in multiple locations, such as the kernel, and runtime libraries. In order to effect runtime change, without restarting the system, a management agent must update all locations using different APIs. Similarly, the persisted hostname can be stored in multiple locations such as distribution-specific configuration files, and local host-address mappings. An SNMP agent must therefore track the configuration interfaces and repositories of each different version of popular Linux distributions. For example, the most current version of the most popular Linux SNMP agent[30] will effect a partial runtime host name change, without updating persistent storage. As a result, a Linux system whose name has been changed through SNMP will report

inconsistent naming information, and will revert to its previous state when it is restarted.

The P2P architecture eliminates the need for management agents and thus improves service *reliability* by reducing the size and complexity of implementing managed network services. The P2P Modeler is the only authoritative repository of management information. Elements and managers use a simple transactional interface to access the shared repository, eliminating the possibility of inconsistencies between runtime and persistent state. Consistency of information that is replicated in the management model can be automatically maintained using synchronous event handlers. Reliability is further improved through consistent element configuration views, and atomic element configuration updates.

2.4 Scaling Network Monitoring and Control

The P2P architecture offers distinct scalability advantages over the MA architecture. Management information is stored in a distributed modeler supporting concurrency control mechanisms. The database approach to management creates a safe and reliable environment for executing concurrent management operations. Management functions can thus be distributed across management servers, or even placed within the elements to effect autonomic behavior. Change management is supported through a publish-subscribe mechanism on all types of management changes. Remote polling is no longer required thereby reducing network and processing load. Event filtering can be pushed to the source of the change events, further reducing load. The P2P management architecture also supports scalable cross-domain configuration management. Domain peering-point information is summarized as a view over the domain model. The Modeler transaction mechanisms assure view consistency, thereby enabling remote domain management.

2.4.1 Efficient Monitoring

Management operations in the MA architecture are centralized into management stations. Centralization is dictated by the need to correlate information for discovery and the absence of manager synchronization capabilities. This centralization of management operations limits the type and number of management functions to the performance of the centralized server. The polling mechanisms[31] employed by MA architecture managers further limit the scalability of management stations in terms of polling frequency, and network and processing overhead. As a result, the management of current networks does not scale in the number of elements, or the types of management operations that can be handled. Network growth is handled today through reductions in polling frequency, and splitting of networks into separate administrative domains.

The P2P management architecture provides *scalable* monitoring and control of network elements. Management events eliminate the need to poll resources, thereby reducing processing and communications load. Management functions can be safely and reliably distributed to multiple managers, and autonomic elements, supporting the Management by Delegation (MbD)[32] paradigm. The MbD paradigm places data-intensive management functions within the network element, resulting in lower polling or event message transmission and processing overheads.

2.4.2 Sharing Discovered Information

Traditional management architectures define Management Information Bases (MIBs) that are restricted to representing the configuration of individual elements. This approach creates management "data islands" which cannot be easily navigated since traditional managers do not export views of their centralized global view of network configuration. Network elements are therefore severely restricted in their ability to discover the topology of their environment. Discovery is a prerequisite for effecting

self-configuration, healing, and optimization.

Relationships in the P2P model can be used to express a unified management schema. Autonomic elements navigate the unified model by using relationships that bridge the physical distribution of data. The unified model may be enriched by managers with discovered information in the form of schema extensions. The transactional access mechanisms of the P2P Modeler support these capabilities in a safe and reliable manner. For example, a link-layer topology manager may add objects representing link-layer broadcast domains and establish relationships to the participating link interfaces. Other managers can utilize this information to propagate changes over the broadcast relationships.

2.4.3 Cross-Domain Management

Domains are used to scale network management, and establish boundaries on network dependencies for the purpose protecting access to sensitive configuration information. A domain is "A list of systems with an organizational boundary or some other extent to which management functions might want to be restricted"[33]. The ability to control the scope of propagated change in a network is a requirement for stable and scalable operation. Unbounded propagation can result in long, difficult to detect, propagation cycles that can cause network instability. Effecting consistency in large networks requires synchronization over large sets of elements which can severely restrict the rate of change, and can result in starvation of change processes.

Current MA architecture-managed networks cannot support safe cross domain change propagation and control. As was previously discussed, the MA architecture does not support safe multi-manager access. Domain control is therefore centralized by function into a individual management stations, and exporting control to other domains is not possible. Cross-domain management is thus restricted

to performance monitoring. As a result, network services are designed with the goal of minimizing cross-domain configuration dependencies. Domains boundaries are established at the network layer, controlled by specialized dynamic configuration protocols such as OSPF and BGP in ways that are not always successful[34, 35]. Furthermore, the increasing virtualization and mobility of network services creates new complex cross-domain configuration and performance dependencies.

The P2P architecture supports scalable and safe propagation of changes across domains. Domains may export summarized views that are consistently synchronized with the domain element management state, using the Modeler transaction mechanisms. Elements management may be shared by multiple domains due to the multi-manager P2P architecture capabilities. Transaction logs may be analyzed to determine cross-domain propagation paths, and to perform synchronous shunting of propagation. Cross-domain management is presented in detail in chapter 4.

2.5 Enforcing Network Policy

The Management Information Bases (MIBs) of traditional management architectures are limited to the specification of management attribute data types. Semantic model information is not expressed since it would require adoption of a standard language for expressing schema semantics, and a mechanism for navigating across MIBs in order to express the semantics of cross-element dependencies. As a result, the semantics of current element configuration are embedded in the element code and are restricted to enforcing local configuration constraints, without any mechanism for notifying managers of aborted updates.

2.5.1 Expressing Semantic Information

The P2P architecture supports an extensible mechanism for associating *semantic information* with the management model. Model semantic information may be expressed in a declarative fashion, supporting analysis of the effects of configuration changes across multiple elements, prior to committing the transaction. The P2P architecture enables the extensibility of the object-relationship schema, without mandating use of a single semantic declaration language. A plug-in mechanism is used to support multiple mechanisms for expressing semantic constraints and change propagation rules. P2P policy enforcement is presented in detail in chapter 4.

2.6 Related Work

The P2P model builds on earlier efforts in the use of object-oriented models to support operations management has been pursued by others. The OSI CMIP proposal was based on Object-Oriented (OO) models to organize instrumentation of managed resources at agents. Various research projects[11, 32, 36, 37] and some commercial products (SMARTS InCharge, HP OpenView, Tivoli TME) have used OO resource models successfully to simplify the development of management applications. This work broadens this effort to build on modeling technologies that can create unifying heterogeneous configuration information directory structures to support automated management. The semantic model captures detailed configuration needed to build self- management/organization software[38].

The proposed architecture is also related to recent work on directory services. This work has traditionally focused on directories of high-level objects, such as documents and files. More recently, the advantage of centralizing management information in a unified schema has led to the creation of a standardized infor-

mation model, initially pursued by the ad-hoc group on Directory Enabled Networks (DEN)[15] and more recently by the DMTF (Distributed Management Task Force)[39]. Future NESTOR versions will support Meta Object Facility (MOF)[16] import/export functions, to assist in leveraging the standards work.

The most closely related management architecture is the ICON system[40] which uses the active database style Event-Condition-Action (ECA) rules to state restrictions on objects instrumented by SNMP MIB values. Both systems borrow ideas from active-database management systems (ADBMS)[41]. The P2P architecture extends these approaches to define a two-layered approach which unifies the roles of managers and elements in a single layer. The P2P architecture incorporates multi-protocol access to heterogeneous resource information, configuration transactions, declarative constraints, and constraint propagation through policy scripts.

The Dolphin project[14] developed a declarative language for modeling network configuration and operation for fault analysis. Emphasis was placed on deducing the cause of failures after the fact, by verifying the propagation of operational rules in the model. The P2P architecture provides mechanisms for preventing failed configurations before they are applied through the use of synchronous event notifications, and transaction rollback.

In the area of configuration management automation, the GeNUAdmin[42] system is an off-line tool for extracting network configuration information into a centralized database, performing updates on that database which are checked for consistency, and pushing the changes back into their respective configuration files. Simple consistency checks are performed to assure that added values are valid and that key values are unique. The RPI service dependency tool[43] detects service dependencies and generates up to date server listings. The goal of the system is to prevent unforeseen service interruptions caused by hidden service dependencies. The P2P architecture can support this functionality given an appropriate set

of constraints on the unified configuration model. Ganymede[44] is an extensible and customizable directory management framework applied to the central management of user and host data, which is distributed in different databases. Ganymede supports transactions on the central repository objects, but does not provide a constraint mechanism beyond a few built-in security, and deletion propagation checks.

The Constraint Satisfaction Problem (CSP) has been studied extensively in a variety of applications[45, 46]. Previous work on constraint-based management has been pursued[47, 48]. The focus of these projects has been on employing constraints for the diagnosis of network faults and on algorithms for constraint satisfaction. The P2P architecture supports this approach by providing a rich model and a safe modeling environment. In the P2P architecture, a CSP engine is represented as a knowledge module.

Simple scripting solutions to network configuration automation are dependent on network topology and the particulars of element configuration mechanisms that differ across vendors and even between versions of the same platform. A single change in network topology or equipment upgrade may necessitate changes in multiple scripts. For these reasons, scripts cannot be easily shared among different installations without significant customization. Errors in script execution can result in inconsistent network configuration states, from which it is difficult to recover manually. It is hard in the context of traditional scripts to enforce exclusive access to configuration repositories. In addition, automatic discovery of relationships not directly instrumented is not practical. The P@P architecture supports the safe use of scripts through the binding of the DAP to libraries for popular interpreted languages, such as Perl[49].

2.7 Summary

The manual process with which computer networks are currently managed is quickly reaching its limits as networks enlarge, add new mission-critical services, and spread to new environments such as private homes. Network management automation is increasingly becoming a requirement in many different types of networks. Large networks are becoming too complex to manage; mission critical networks cannot afford operator errors; and small home networks must minimize management due to limited resources. Current practices will become unmanageable in future networks supporting autonomic reconfiguration and programmability for service deployment. The P2P architecture addresses these needs by combining several techniques from object modeling, constraint systems, active databases, and distributed systems in novel management architecture.

The P2P architecture is based on a two-layer approach to management. Control of management configuration and performance information is transfered from the network elements into a distributed object modeler. Management information is represented using objects and relationships in a unified model. All access to the distributed management modeler is controlled by a transactional interface. Thus, the traditional roles of elements and managers are unified over this common interface. Elements access the model to export their state, read their configuration, and discover their environment. Managers access the model to discover element topology and effect configuration change. The P2P architecture provides as safe and reliable environment for autonomic configuration. The architecture supports scalable network monitoring and service discovery through a rich event mechanism over the unified model. Management data models can be extended with semantic information using synchronous event handlers. Asynchronous events allow monitors to correlate network events in terms of their triggering transactions. These properties will be analyzed in detail in the following chapters.

Chapter 3

A Language for Autonomy

3.1 Introduction

This chapter introduces a novel approach to developing services with embedded autonomic configuration capabilities using a language called JSpoon. In order to effect autonomic behavior, a service must instrument its operational behavior and external interactions with other services. It needs to represent this information in a model which admits automated interpretation and control, incorporating knowledge on how to automate management actions. JSpoon extends the Java language with appropriate management attribute declarations, management synchronization primitives, and event declarations. Runtime features support management persistence, and dynamic schema semantic extensions in the form of knowledge plug-ins.

JSpoon service objects are extended at design-time element with specialized JSpoon management attributes representing configuration and performance information. The JSpoon attributes of a class define the management schema of that class and may be accessed by the service itself, or by other JSpoon programs. This approach essentially unifies the traditional management roles of element, agent, and manager under a common data model layer.

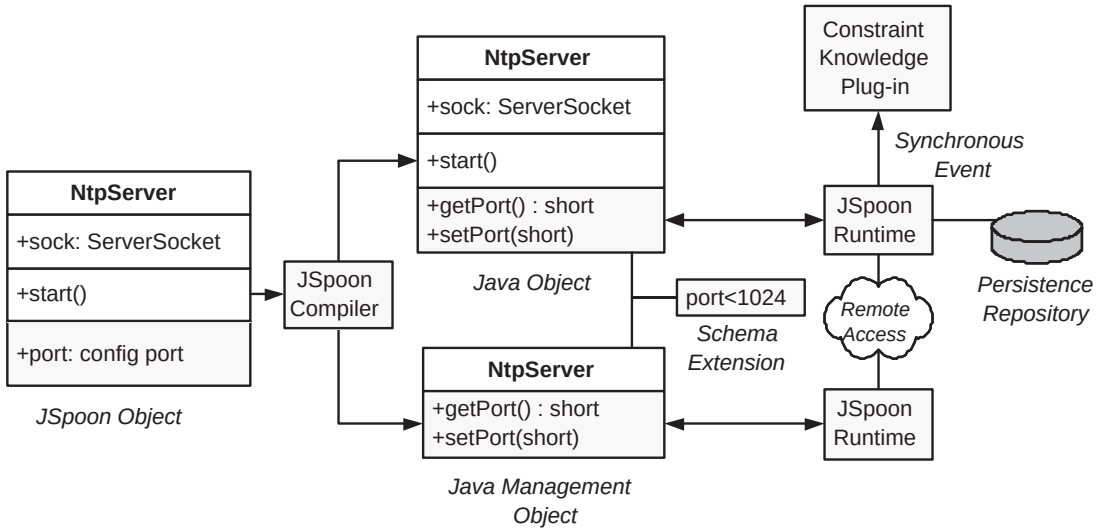

Figure 3.1: JSpoon Feature Overview

JSpoon programs access management attributes in a transactional manner. The management type system restricts the operations permitted on management attributes. For example, a performance counter attribute may only be incremented. JSpoon services may subscribe for configuration events in a synchronous or asynchronous manner. Event notification occurs in the context of the originating transaction, allowing subscribers to effect additional changes, or request a rollback.

The JSpoon management schema can be extended at runtime with additional "meta" attributes, new management functions, as well as semantic information. Semantic knowledge is introduced as an extension to the data model schema in the form of plug-in modules. The plug-ins perform tasks such as constraint verification, change propagation, and fault-analysis. In this manner, the knowledge needed for autonomic behavior can be independently created and incorporated.

Consider a simple network service such as a Network Time Protocol (NTP) service [50]. A common Java-based implementation would represent the service as a Java object encapsulating a network server socket object bound to a specific

port, and a relationship to a time source. Without significant additional effort from the object programmer, attributes such as the service port or network socket performance will not be manageable. JSpoon enables programmers to design configuration and performance attributes as an integral part of service object class design using minimal effort.

The key features of JSpoon are depicted in figure 3.1. The example shows an NTP service UML class diagram whose class model is extended with a JSpoon management section. The class contains regular Java class attributes and methods, such as a server socket and a `start()` method, as well as configuration port management property. The JSpoon object is compiled into a Java class with the non-management Java attributes and methods, as well as accessor methods to the JSpoon configuration attributes. An additional management class is introduced which exports the management attributes through accessor methods of other JSpoon programs. At runtime, programs accessing configuration attributes interact with the JSpoon runtime environment which provides persistence, concurrency control, remote access, and event notification services. The management schema of JSpoon services may be extended dynamically with semantic information. For example, a constraint may be associated with the port number, as shown in the figure. The JSpoon runtime will invoke the appropriate plug-in to evaluate the expression after the value has been changed.

Although JSpoon is introduced as a Java extension, its features are equally applicable to other modern object-oriented languages, such as C#. Additional language bindings are left as future work.

3.1.1 Summary of Results

JSpoon simplifies *service development* by automating the export of management attributes. Developers are thus encouraged to expose all configuration attributes

resulting in an overall improvement in *service manageability*. Automation further reduces the need for programmers to deal with multiple copies of a configuration attributes, for internal, persistence and standard MIB support, thereby improving *safety*. Transactional access to configuration properties creates a safe environment for *concurrent management* of network services. JSpoon relationship declarations enable the linking of related services creating a mechanism for service *discovery*.

The JSpoon management events extend Java with a new *dynamic service extension* capability. Synchronous events are a mechanism for extending the behavior of services to provide additional change propagation, and configuration verification capabilities. This capability *generalizes exception generation and handling* from a single thread of execution to a set of distributed processes. Asynchronous management events provide improved *fault root cause analysis* based on the logging of changes propagating across multiple elements.

JSpoon schema plug-ins support runtime *data and semantic extensibility*. Access to management information is necessary in a variety of applications, such as configuration automation, performance and fault monitoring, and inventory control. Information collected from JSpoon elements can be used to dynamically *enhance the configuration model*. Semantic schema extensions can provide *protection* from inoperable, or inefficient configurations based on the enhanced configuration model. Service behavior may be *customized* to local operational requirements through change propagation plug-ins. This approach creates *new markets* for systems services for plug-in vendors, and plug-in service providers.

While there have been other attempts at providing management library support to Java-based services[51], JSpoon is unique in supporting management at the language level. The language-level integration simplifies development by automating management schema and accessor generation, thereby increasing code reliability. The syntactic nature of the synchronization and event subscription features creates

simpler semantics, and enables additional optimizations.

3.1.2 Manageability Challenges

Network services are typically developed in general purpose programming languages, such as C, C++, and Java, using standard Integrated Development Environment (IDE) tools. Despite the proliferation of standard and third-party libraries providing common data structures, graphics, and communications abstractions, there has been little or no standardization on management design patterns and programming interfaces (APIs). As a result, system designers and implementors are forced to design their own internal configuration abstractions, persistence mappings, and remote access protocols. This process typically requires designers to:

1. Identify each *program decision point* which supports customization. For example, a program statement opening a network server socket connection.

2. *Represent* each configuration value as a program or class variable. In the server socket example, this would be the port number to be used.

3. *Persist* configuration settings:

   (a) Select a *persistent storage repository*. This is typically a file, but may also be a platform specific repository such as Microsoft Windows Repository.

   (b) Define the *service configuration schema*. In the past, flat type-value pairs were typically used, but hierarchical structures are increasingly being adopted in the form of XML schemata.

   (c) *Program a parser* for retrieving and validating values from the repository. With the advent of XML this step has been significantly simplified,

but the mapping the hierarchical schema into the class model is not an automated process.

(d) Create a mechanism for *user configuration access*. For configurations stored as a file, the interface may be a text editor, or a custom graphical user interface. For XML-based persistence, XML editors could be employed, but current editors have not demonstrated ease of use.

(e) If the service effects changes on its own persistent configuration, *program a generator* for dumping the current service configuration into the repository. If the repository does not support concurrency control, configuration may be corrupted due to concurrent external changes.

4. *Validate* the value for each restored configuration attribute prior to use.

5. *Monitor* the persistent repository for changes, and and safely update the runtime state. This can be a challenging requirement since monitoring occurs on a background thread that must synchronize with service threads to effect safe configuration state updates. In the network port example, a change in port configuration will require that the service close the previous server socket, and open a new one. In practice, most services do not support dynamic configuration updates, and therefore must be restarted when their configuration has changed.

As can be observed, manageability adds significant complexity to program development. Because management is not commonly perceived as a core function, services continue to be designed with minimal instrumentation, and simple configuration access and persistence mechanisms. Compliance to standard management schemata is typically added as an additional layer over the native configuration. The layering creates additional data and semantic mapping challenges.

Synchronization is another challenge to configuration system design and implementation. When program behavior depends on multiple configuration attributes, it is important that the program obtain a consistent view of configuration. File repositories may be seen as supporting a primitive atomic operation, but depend on users performing file save operation after completing all relevant configurations. Management protocols, such as SNMP, however, perform individual configuration updates, which can provide inconsistent views to the service and other managers. Services supporting dynamic configuration using specialized protocols, such as dynamic DNS, are particularly vulnerable to race conditions, and must protect access to their internal configuration.

Semantic information, in the form of service operational constraints, is currently spread in volumes of published documentation, user support forums, and personal administrator experience. Deployment of complex services such as database servers, and multi-tiered application servers depends on highly paid experts for setup and optimization. If such constraints are ever expressed programmatically, it is as part of the service program code. Analyzing a general-purpose programming language to extract such constraint information is practically and theoretically impossible. Due to the isolated nature of current configuration repositories, services cannot programmatically express constraints on the configuration of related services.

Current services cannot be dynamically extended with new functionality. Configuration is typically consulted only at start-up time, and therefore adjusting behavior based on configuration changes is a disruptive process requiring service restart. Services supporting dynamic configuration through standard or proprietary management protocols do not support flexible event systems. Adding new functionality requires continuous configuration polling, and introduces configuration consistency issues due to the absence of strong synchronization primitives. As

a result, service users requiring new functionality must rely on requests for improvement submitted to service vendors. Service feature introduction follows a long process of evaluation, design, implementation, testing and release. Requested features may only appear after several long release cycles. Smaller clients may not find recurse through the service vendor's support structure.

The chapter continues with a section covering the JSpoon configuration modeling and synchronization constructs, along with the basic JSpoon runtime persistence, remote access, and discovery services. The following section presents the JSpoon event model and syntax. The knowledge plug-in language mechanism is described in the next section. The chapter concludes with a section on JSpoon compilation issues, a section on related work, and conclusions.

3.2 Embedding Management Functions

This section introduces a novel language-based approach to programming autonomic services with embedded management features. Configuration and performance attributes are distinguished from regular program attributes at program design time. Autonomic services are implemented in a language called JSpoon, which extends Java with declarations of management class and instance variables. JSpoon management variables encapsulate either configuration or a performance values. Configuration attributes are accessed in a transactional manner by and may be read or set by any JSpoon program. Performance attributes represent service state, and can only be set by the object owner. Type modifiers enforce additional management access pattern restrictions. Configuration attributes may be declared as non-persistent, constant, or key values. Performance attributes can be declared as strictly increasing, or computed values. Atomic access to management attributes is supported through a language synchronization primitive.

The JSpoon language runtime is responsible for exporting management ser-

vice information to other JSpoon programs, maintaining management access atomicity, and persisting configuration values. Navigation is supported through relationships between objects instantiated in separate JSpoon services. JSpoon service management information is exported to other JSpoon processes in the form of proxy objects. Atomicity is maintained by a distributed transaction manager implementing the two phase commit protocol[52, 53]. Persistence requires programmer cooperation in establishing unique object identifiers in the form of key attributes, or programmer controlled names.

The next subsections present these JSpoon language and runtime features in detail.

3.2.1 Configuration Modeling

The set of variables declared in JSpoon forms the *management section* of the Java object, which is exported to other JSpoon programs. There are two types of JSpoon variables: *configuration*, and *performance* variables. Configuration variables are used to control program behavior, whereas performance variables instrument program status. Configuration variables may be persisted across Java Virtual Machine (VM) invocations. Access is controlled by the JSpoon runtime environment to support transaction semantics. Atomic access is expressed using the JSpoon language locking constructs. In contrast, performance variables may only be updated by the owner of the object, are not persistent, and may not be accessed transactionally.

Table 3.1 provides an example of a JSpoon object class. The class implements a Simple Network Time Service (SNTP)[50]. SNTP is a UDP-based protocol for querying time servers over an Internet Protocol network. As illustrated in this code fragment, the SNTP service implementation is written in standard Java extended with JSpoon declarations. The class encapsulates a regular Java instance variable called `sock` which maintains the server's UDP binding. In addition, three

```
public class NtpServer extends Thread {
  protected DatagramSocket sock;

  config key int port = 123;
  config boolean active = true;
  instrument counter long reqCount = 0;

  public NtpServer() throws ... {
    sock = new DatagramSocket(port);
  }

  public void run() {
    while(active) {
      sock.receive(packet);
      reqCount++;
      // process request ...
    }
  }
}
```

Table 3.1: JSpoon Autonomic Service Class Example

management instance variables are declared in the extended syntax of JSpoon: `port` and `active` (configuration), as well as `reqCount` (performance).

The `port` instance variable stores the NTP service UDP port number. It is a configuration variable, as identified by the JSpoon modifier `config`, of primitive type `int` initialized to the default SNTP UDP port 123. The `key` variable states that the value must be unique in all instances of the class within the Java VM. The second configuration instance variable, `active`, controls termination of the server process. The `reqCount` performance variable, as identified by the `instrument` modifier, counts the number of time queries received.

From the service's perspective, the declaration and usage of JSpoon management variables is similar to that of regular Java variables. There are two main differences: (1) management variables are exported through the JSpoon runtime environment to external management processes, and (2) the types of operations

Modifier	To	Description
`computed`	i	Evaluate on-demand; not stored
`counter`	i	Monotonically increasing value
`final`	c	Assign-once (Java semantics)
`key`	c	Unique object identifier
`static`	c, i	Property of class; not instance
`transient`	c	Non-persistent variable

Table 3.2: JSpoon Declaration Modifiers

permitted on the management variables may be restricted based on the JSpoon modifiers. For example, an external manager may change the value of the `active` configuration property, thereby effecting termination of the service. Similarly, the operations allowed on the `reqCount` *counter* performance variable are restricted to monotonically increasing value updates.

JSpoon modifiers may only be applied to class and instance variable declarations. The JSpoon compiler will generate a syntax error if JSpoon modifiers are used in other contexts, such as in method argument or variable declarations.

Modifiers

JSpoon configuration and performance variable declarations may be specialized with the modifiers listed in Table 3.2.

Computed variables have their value evaluated on-demand using a Java expression. For example, the computed free memory performance variable shown below is associated with a Java expression for obtaining the free memory in the Java VM.

```
instrument computed long freeMemory =
  Runtime.getRuntime().freeMemory();
```

Computed variable expressions are bound to the scope of their declaration.

Numeric performance JSpoon variables marked with `counter` are restricted to a monotonically increasing value. The counter marker may be applied to the primitive number types, and objects implementing the `java.lang.Comparable` interface. The only operations permitted on primitive numeric types are `++` (increment by one), and `+=` (increment by value, where value ≥ 0). Numeric object types are immutable, therefore the only allowed operation is assignment (`=`) where the assigned value compares-to greater, or equal to the previous value.

Configuration variables may be marked as `final` to specify that they may only be assigned once and cannot not be modified by the program or an external entity. It is possible to change persistent final variables in between program invocations. Typically, variables will be declared final if the program is not able to adjust its behavior after startup. For example, the user ID under which a Java process should be executed may be declared as a final variable.

Certain configuration variables may uniquely identify an object instance. For example, in the SNTP UDP server of table 3.1, the port variable is defined as a `key` variable since only a single instance of `NtpServer` can bind to the same unicast UDP port. When multiple configuration variables are declared as keys their combination uniquely identifies the object.

The `static` JSpoon modifier associates its configuration or performance variable with the enclosing class, not an instance as is the default.

Configuration variables may be marked `transient` to indicate that the JSpoon runtime should not maintain their values across Java VM invocations, or when the object is serialized. Persistence requires additional changes in the way Java programs manage the life-cycle of their objects, and is covered in the next section. By default, all configuration variables are persistent. Instrument variables are always transient, and can only be persisted through explicit assignment to a configuration variable.

```
enum Status { stopped, running };
instrument Status status;

public void method() {
  status = Status.stopped;
  // ...
  if (status == Status.running) ...
}
```

Table 3.3: Enumeration Type Example

Unlike Java class and instance variables, JSpoon variables are not associated with an access modifier such as `public`, `protected`, or `private`. All JSpoon variables are publicly accessible via the JSpoon runtime. A role-based security policy may be configured at the modeler-layer, or through a security knowledge plug-in, and will not be presented in this paper.

Types

JSpoon variables are strongly typed in the Java type system. All Java primitive types are allowed in JSpoon variable declarations. Java object types may be used if, and only if, they represent immutable objects, and are serializable. Since here is no Java marker interface identifying objects as immutable, the JSpoon compiler includes a list of immutable standard Java library object classes. User-defined classes employed in JSpoon variable declarations must implement the `jspoon.Immutable` marker interface and all their non-JSpoon variables must be declared as `final`.

JSpoon extends the Java type declaration system with support for enumeration types. The example in table 3.3 shows an enumeration declaration of the type `Status` with two enumerated values. An enumeration type is *conceptually* mapped into a Java inner class of the same name, with immutable instances for each enumeration value. These instances are available as static final variables of the enumeration class. As shown in the example, an enumeration variable may be

```
public class HttpServer
  extends Thread {
  relationshipset threads, HttpThread, serves;
  public void run() {
    while(true) {
      Socket s = sock.accept();
      threads.add(new HttpThread(s));
    }
  }
}
public class HttpThread
  extends Thread {
  relationship serves, HttpServer, threads;
  public void run() {
     // Process HTTP request ...
     serves = null;
  }
}
```

Table 3.4: Relationship Type Example

assigned or compared for reference equality with the static enumeration instances.

JSpoon relationship declarations are used to establish an association between two Java objects. Unlike simple references, relationships can be navigated in both directions. Each end-point in the binary association is associated with a role (variable identifier), and multiplicity (to-one, to-many-set, or to-many-sequence).

A sample relationship declaration is shown in table 3.4. Instances of the class `HttpServer` are associated with a set of `HttpThread` objects through the `threads` variable role. The same relationship must also be declared in the `HttpThread` class, with the role names reversed. In this case, the `serves` variable role is declared with a to-one multiplicity. Variables identifying to-one relationship roles may be used as normal Java object reference variable. To-many relationship variables support accessor methods for retrieving, adding, removing and setting their membership.

3.2.2 Controlling Management Access

An autonomic element coded in JSpoon will contain classes with a mixture of management and standard Java attributes. The management attributes, as presented in the last section, are characterized by restrictions on the types of permitted access. Because configuration attributes control behavior, it is important to maintain consistent views of their values. Consistency may be violated by the autonomic element itself, if its multi-threaded, or other autonomic elements attempting to reconfigure the element. For example, a service may need to access its configuration to obtain the address of a related service, and the necessary credentials for accessing that service. The access to these two configuration variables should be made atomic in respect to configuration changes.

Another important consideration in management involves the ability to recover from failed configuration changes. Changes may occur through direct user intervention, or through automatic propagation of changes. For example, a change in a router's configuration may also propagate through a dynamic routing protocol, such as OSPF. In the absence of transactional information, identifying and recovering from such fragmented changes requires application of event-correlation methods whose inferences may be incorrect.

JSpoon addresses the aforementioned issues by enforcing transactional access on all configuration attributes. Transactions may be created explicitly, using an `atomic` block which is associated with the executing thread, or implicitly, by accessing the value of a configuration attribute. Transactional operations may cross repository boundaries when programs navigate relationships, and the JSpoon runtime maintains transaction properties across process and machine boundaries. JSpoon program state synchronization is performed implicitly every time a configuration attribute is accessed, or may be performed explicitly using the event mechanisms that will be covered in the next section.

```
atomic(lock-timeout) {
  z = x + y;
}
```

Table 3.5: Atomic Action Example

Atomic Operations

JSpoon expresses concurrency control similarly to the way that Java handles thread synchronization on object methods and variables. JSpoon programs requiring atomic access to multiple configuration attributes must perform these accesses within an `atomic` block. The atomic block provides JSpoon programs with atomicity, consistency, isolation, and durability (for persistent properties) in accessing management attributes.

An atomic block example is listed in Table 3.5. The JSpoon program encloses its access to the configuration properties `x`,`y`, and `z` in a `atomic` block to assure consistent change. The JSpoon runtime will generate the required read lock requests for `x` and `y`, as well as a write lock request for `z`.

The obtained locks are owned by the thread of execution. Nested transactions are supported through syntactically nested blocks, or through invocation of methods containing atomic blocks. The effected changes are committed at the end of the block, and the acquired locks are released, unless this is a nested transaction. If an exception is thrown from within an atomic block, then the effects of the transaction are rolled back.

Lock requests may fail due to a communications failure, detection of a deadlock, or lock acquisition timeout. These errors are signaled in the form of exceptions. If an exception is thrown in an atomic block, then the values of all *management* configuration attributes will be restored to their previous values. The syntax of the atomic block supports the optional specification of a lock acquisition timeout.

Atomic blocks may also be used to group the generation of performance

variable update events. Performance variable updates performed within an atomic block do not generate modeler update events until the end of the block. This mechanism may be used to provide consistent views of performance variables and to reduce the overhead of synchronization between the service thread and the JSpoon event monitoring thread.

There are several advantages to expressing atomicity as a language feature, as opposed to an API feature. The syntactic nature of `atomic` blocks permits compile-time checking for transaction termination. Atomic blocks may be analyzed at compile time to perform lock-request ordering optimizations which can reduce and some time prevent transaction deadlocks. Implicit transaction composition in the form of nested transactions does not require explicit programmer management.

3.2.3 Management Services

A configurable service typically needs to persist its configuration between invocations. This persistent configuration must be made available to administrators for editing and customization. Service authors must therefore define a management schema, provide code for dumping the current, and for parsing stored configuration state, as well as performing sanity checks on restored configuration states. In many cases, custom graphical user interfaces must also be developed to improve ease of use, and provide monitoring capabilities.

Another common service requirement is the export of process performance and status information, such as CPU, memory and network utilization. Service providers must instrument their programs with performance monitors and map these values into their management schemata. Such instrumentation code can obscure the main service code, and can also effect performance through calculation of performance attributes that are not needed.

Services depending on other services and wishing to adapt their configuration

accordingly must obtain access to the other service's configuration. Such access may involve parsing of configuration files which is onerous on development, and creates version dependencies between services. If the service must reconfigure its peer service, this problem is exacerbated, and introduces other issues such as security, and access control.

JSpoon addresses the above issues by providing a set of management runtime services. Configuration attributes may be persisted automatically with minimal effort from the service author. The unified nature of the configuration repository enables the creation of a single graphical user interface for viewing and editing configuration, as well as for performance monitoring and visualization. Common management monitoring and configuration requirements are handled through a set of managed library objects which encapsulate standard Java objects, such as network sockets and file streams, with a rich set of properties. The JSpoon runtime performs lazy evaluation of such measurements based on demand. JSpoon also provides remote repository access to enable navigation of relationships that cross process and machine boundaries. The unified model erases the difference between local and remote configuration access.

Persistence

Modern programming language environments provide multiple mechanisms for persistence, such as, file access, relational database connectivity, and directory accesses. Most options require programmers to explicitly manage the bindings between the persistent storage schema and the implementation class schema. Synchronization between persistent and runtime state also needs to be explicitly managed through identification of state synchronization points.

The JSpoon runtime environment supports automatic management of object configuration variable persistence. In order to enable persistence, classes must

implement the `Persistent` marker interface. Persistent objects must be assigned a unique *object identifier* (OID) in order to support retrieval when instantiated within the Java program. Objects which have key variables, all of which are final and persistent, can be automatically assigned a unique OID. In all other cases, the programmer is responsible for assigning the unique OID.

The JSpoon compiler modifies the signature of persistent class constructors to include an additional argument of type `jspoon.JSpoonOID` and to throw the `PersistenceException` exception. The constructors are also modified to include the necessary hooks for retrieving previously persistent variable values. It should be noted that the persistent values will override any default variable values specified in the declaration section. At construction time, the programmer must provide the additional OID parameter to establish the unique identity of the object.

Table 3.6 shows the `NtpServer` class after it has been marked to support persistence, and a simple simple `TimeDaemon` application which uses a single instance of the `NtpServer` class. At construction time, the additional OID argument must be provided to establish the object's identity. A static identifier can used is this case because the application is limited to creating a single instance of the persistent class.

Every JSpoon process must be assigned a unique *service identity* (SID). Service IDs are required in order to prevent multiple instances of a program from owning the same persistence repository data. Persistence repositories must support locking to prevent concurrent binding of multiple JSpoon programs using the same SID. A lease-based mechanism is employed to support releasing of resources following a service failure.

A fully qualified object ID contains the service ID as well as the location of its persistent repository. It is possible to set the default service ID and repository location of a JSpoon process in order to simplify construction of OID objects.

```
public class NtpServer extends Thread
  implements jspoon.Persistent {
  // ...
}
public class TimeDaemon {
  public static void main(...) {
     JSpoonOID oid = new JSpoonOID("jspoon:NtpServer#Singleton");

     NtpServer srv = new NtpServer(oid);
  }
}
```

Table 3.6: Persistence Example

Services can have multiple SIDs and connect to multiple persistence repositories.

An object ID may be represented as a URI[54] of the form:

`jspoon://`*userinfo*`@`*host*`:`*port*`/`*serviceID*`#`*OID*

For persistent objects with final key attributes the OID may be expressed as:

className`?`*key1*`=`*value1*`,`*key2*`=`*value2* ...

The Java language supports objects serialization as a mechanism for storing and transmitting object state. The JSpoon compiler modifies the `writeObject` and `readObject` methods of serializable objects to include marshaling and unmarshaling of variables in the management section, and code for binding the deserialized object into the local JSpoon runtime environment.

Remote Access & Discovery

JSpoon programs requiring access to management attributes of remote objects must first obtain a local copy (view). This is performed by using the static methods of the `jspoon.JSpoonNaming` class. The `list` method supports query of objects based on a query URI. In the example of table 3.7 the manager requests the OIDs of all instances of the `NtpServer` class. The example code assumes that at least one

```
public class NtpMonitor {
  public static void main(...) throws ... {
    JSpoonOID[] oids =
      jspoon.JSpoonNaming.list("jspoon://localhost/NtpServer");
    NtpServer server =
       (NtpServer) jspoon.JSpoonNaming.lookup (oids[0]);
    while(true) {
      System.out.println(server.reqCount);
      Thread.sleep(5000);
    }
  }
}
```

Table 3.7: Manager Example

OID is returned, and then invokes the lookup method to obtain a local view of the JSpoon object. Subsequently, the JSpoon object can be accessed in the manner illustrated in previous examples.

If the listing and lookup methods are enclosed in an atomic block, the JSpoon runtime will obtain appropriate locks to assure that the results remain stable. In the listed example, had the listing and lookup methods been enclosed in an atomic block, the runtime would lock `NtpServer` object creation and remove effects.

Managed Object Library

The JSpoon environment provides managed versions of standard Java library objects. JSpoon managed objects follow a naming convention of appending the jspoon prefix to the full class name of the instrumented class. In this manner, the `jspoon.net.JSpoonDatagramSocket` class supports configuration and performance instrumentation of UDP datagram sockets. Use of managed objects can greatly increase the management flexibility of Java programs. For example, use of the managed socket class in the `NtpServer` would enable managers to configure socket options such as the traffic class.

The JSpoon managed object library also includes mutable versions of immutable Java objects by encapsulating reconfiguration semantics. The resulting manageable objects can be used by applications requiring reconfiguration.

3.3 Reacting to Autonomic Changes

Network configuration changes due to new element deployment or failure of existing element. An changed element may be a logical service, such as a web-server, or a physical service, such as a link. Network performance changes in response to request patterns, as well as configuration changes. The operation of autonomic services may be effected by such changes, and require further configuration changes in order to maintain availability, performance, or security requirements. Therefore, a service autonomic environment must provide a change notification mechanism.

Existing management systems, such as SNMP, offer restricted event mechanisms. The types of monitored events are predefined as part of the management schema (MIB), and typically represent changes in performance, as opposed to configuration. Configuration management systems are therefore required to poll elements for configuration changes. Configuration changes discovered through either notification or polling have already effected service behavior. As a result, operational violations cannot be prevented by external managers. Misconfiguration can result in complete service failure in a manner that is not recoverable without physical access. Configurations violating security can create periods of vulnerability which may be exploited.

JSpoon supports automatic generation of low-level management events enabling JSpoon managers to perform asynchronous as well as synchronous event handling. All JSpoon management accesses are treated as events which can be exported. Complex events may be defined in terms of the basic JSpoon events. Synchronous event handling occurs in the context of the triggering transaction.

The combination of flexible event definition, with synchronous handling creates a powerful mechanism for dynamically extending service management context.

3.3.1 Identifying Autonomic Events

JSpoon programs may subscribe to events on configuration and performance management attributes. Primitive JSpoon events are triggered by transactional updates to configuration properties of management objects, and non-transactional updates to performance attributes. Conditional events filter notification of primitive events based on a conditional statement over the changed object. Event notification may also be temporally conditioned on over event ordering within a transaction. Performance events express a monitoring condition on the value of a JSpoon instrumentation management attribute.

Primitive Events

A primitive event E is defined by the tuple <*type*, *transaction*, *time*, *target*, *value*, *oldValue*, *index*> where:

- *type* is one of the seven primitive types: class `load`, class `unload`, object `create`, object `delete`, attribute `set`, relationship `add`, and relationship `remove`.
- *transaction* is a reference to the transaction in whose context the event occurred,
- *time* is represented as an offset to the start of the transaction. Time does not define a total ordering since the transaction may be shared by concurrent threads executing on different processors, or hosts. Event ordering was covered in the previous chapter,

```
// Attribute events
lease = subscribe NtpServer.port // ...
// Instance attribute events
lease = subscribe srv.port // ...
// Class load/unload events
lease = subscribe NtpServer // ...
// Any attribute event
lease = subscribe NtpServer.ALL_FIELDS // ...
```

Table 3.8: Primitive Event Subscription Examples

- *target* is the field which was set (for attribute set, and relationship add/remove events), or the class (for class load/unload and object create/delete events),
- *value* is the field value or object instance created/removed,
- *oldValue* is the previous value of the field,
- *index* is used in ordered relationship additions to indicate the index of the value added.

JSpoon supports event subscriptions as a language construct. The `subscribe` keyword takes an event condition expression as its first argument followed by an action argument. Subscriptions return a JSpoon lease object used to manage their life-cycle. The action part of event subscriptions will be covered in the next subsection.

Table 3.8 lists four primitive event subscription examples. The first subscription matches assignments to the *port* attribute on all `NtpServer` class instances. The second subscription example operates similarly, but on a specific class instance. If the instance is deleted, the associated subscription leases will be canceled. The third example requests notification of instance `create`/`delete` on the `NtpServer` class. It is possible to subscribe to all class attribute or relationship events using the `ALL_FIELDS` marker.

```
subscribe NtpServer.port on port != 123 // ...

subscribe NtpServer on port != 123 // ...
```

Table 3.9: Conditional Event Subscription Example

Conditional Configuration Events

Conditional event subscriptions allow primitive event subscriptions to be filtered by a boolean expression. For example, the `NtpServer port` subscription handler shown in the previous example may only need to execute when a non-standard port number is assigned. While it is possible to perform the check within the handler, expressing the restriction as part of the event subscription can allow the JSpoon environment to optimize event generation. This is particularly important when the event subscription operates over remote objects or performance attributes.

The first example in table3.9 shows a subscription on `port` assignments to a non-standard port. The second example operates identically to the first, but depends on the JSpoon compiler to determine the triggering attributes or relationships by analyzing the conditional expression.

Temporal Configuration Events

The transaction context creates a natural grouping and ordering of primitive events. JSpoon supports temporal event subscriptions. Temporal event subscriptions allow users to subscribe to events based on the existence or, or absence of a previous event in the transaction history. The `follows` keyword can be used to declare a comma separated list of one or more events that must precede the subscribed event. For example, a JSpoon program may subscribe for `port` attribute events on an `NtpServer` instance only if they are preceded by a in the `NtpServer.enabled` attribute of the same object, as shown in the first statement of table 3.10.

Temporal events can be nested in order to specify a sequence of events.

```
subscribe NtpServer.port follows enabled // ...
subscribe NtpServer.port follows enabled, name follows serves // ...
subscribe NtpServer.port follows port // ...
```

Table 3.10: Temporal Event Subscription Examples

```
Lease lease = monitor NtpServer for utilization > 0.9 over 30000 // ...
```

Table 3.11: Monitoring Event Subscription Examples

Comma separated events may occur in any order. The second example of table 3.10 will match sequences of the form (`port` ... `enabled` ... `name` ... `servers`), or (`port` ... `name` ... `enabled` ... `servers`). The third example of the same table demonstrates how the temporal construct can be used to detect cyclical changes of properties, by subscribing to repeated assignment events over the same property.

Performance Events

Updates to JSpoon `instrument` attributes updates do not occur in a transactional context. The only effect of an `atomic` synchronization construct over performance attributes is to group the notification of updates. The `monitor` keyword allows JSpoon processes to request monitoring of performance attributes. Monitoring requests are expressed on classes, or instances with a boolean expression used to specify the event condition. The `over` keyword may be used to specify a duration for which the expression has been true prior to the firing of the event. The monitoring event handler is invoked upon event occurrence, and will be evaluated *only once* for every change in the monitoring expression value.

Table 3.11 shows a performance event subscription example. The JSpoon performance monitoring subscription will evaluate the handler when the utilization of an `NtpServer` exceeds 90% over 30 seconds. Note that the event handler

will only be invoked once for every period of heavy utilization that exceeds 30 seconds. The performance event handler may update JSpoon object configuration attributes. This mechanism may be used to propagate performance information into the configuration state.

3.3.2 Effecting Autonomic Behavior

An autonomic element must recognize and respond to changes in its environment. Changes may necessitate re-computation of operational parameters to assure service availability and performance. JSpoon autonomic elements discover and monitor their environment by navigating the object-relationship management schema instantiation. Changes in the environment are reflected as changes in the model object topology. Notification of model changes is effected using the event subscription mechanisms described in the first part of this section.

JSpoon supports two types of event handling semantics: *synchronous* and *asynchronous*. Synchronous notification occurs in the context of a transaction, allowing the event handler to perform additional updates, or request a rollback. Synchronous handlers are used in enforcing autonomic configuration policies, as well as extending autonomic element behavior. Asynchronous events are used to monitor system behavior for applications such as fault root cause analysis and intrusion detection.

The JSpoon subscription facility defines a generic Event-Condition-Action (ECA) mechanism, raising the issues of termination and confluence[55, 56]. Using established ECA rule management techniques, the JSpoon compiler analyzes event subscription expressions and generates a dependency graph. This graph is used by the JSpoon environment to identify and monitor possible cycles or order-sensitive rule evaluations. The next chapter will introduce a limited change propagation language over the JSpoon object-relationship model which can be statically analyzed

```
subscribe NtpServer on ( (port < 1024) &&
                         (servedBy.osType == UNIX) &&
                         (user != "root") ) synchronous {
  if (runAsRoot)
    user = "root";
  else
    abort;
}
```

Table 3.12: Synchronous Event-Based Autonomic Configuration

at compile-time far cycle and ambiguity detection.

Extending Autonomic Behavior

Synchronous JSpoon configuration event subscriptions are declared using the `synchronous` modifier, followed by a JSpoon block. A `synchronous` subscription results in the immediate execution of the event handler when the event condition is satisfied within the context of the event triggering transaction. Using this feature, the autonomic capabilities of a JSpoon service can be extended dynamically. Performance events do not occur in the context of a transaction and therefore cannot be accessed synchronously.

For example, consider a Network Time Protocol (NTP) service implemented in JSpoon. The service class will include configuration properties for representing the UDP network port number, as well as the user account under which it should be executed. On Unix systems, only the administrator (root) user is permitted to bind to network ports below 1024. It is possible that the NTP service vendor did not test the service in Unix environments, and therefore failed to express this constraint in the service configuration tool. With JSpoon, this requirement may be later expressed by a third party in the form of a synchronous event handler that changes the service user in response to a privileged port assignment on Unix systems, as listed in table 3.12.

The example subscribes for all changes to `NtpServer` instances whose properties violate the Unix privileged port access condition. The JSpoon compiler optimizes condition evaluation by identifying the specific properties which can satisfy the event condition. The event handler block is evaluated in the context of the effected object. In this example, the handler accesses the `runAsRoot` attribute of the effected `NtpServer` instance and if it is set to `true` configures the service to run as the root user, otherwise it aborts the transaction. The `runAsRoot` property is used to model whether the service is executed as root by the operating system.

The `before` keyword may be used in synchronous event subscriptions instead of the `on` keyword, to effect the handler update prior to application of the triggering change in the transaction log. This feature is useful in situations where the application of committed transaction updates cannot be fully isolated from the environment. For example, a software package installation may require that the software platform installer be upgraded to a specific patch level. Behavior effecting security may also require that an operation such a firewall rule addition precede the installation of some application or the opening of some port.

Using the `event` object reference, it is possible to express conditions on the state of the transaction. For example, constraints that only have to be verified as post-conditions may use `event.transaction.state == VOTING` as a condition.

Generalized Exceptions Synchronous events can be viewed as a generalized exception mechanism. Traditional programming language exception mechanisms are based static declaration of `try-catch` blocks in the program's source code. Exceptions are not exported outside the executing thread context, and the handling procedure is typically bound to a specific code block at compile-time. The exception hierarchy is similarly fixed, depending on explicit application-level exception throwing. The JSpoon event mechanism associates an exception handling point with each configuration or performance attribute modification. Synchronous event

```
atomic {
  // Transaction create event
  // NtpServer.port pre-assignment event (before handler)
  srv.port = 1123;
  // NtpServer.port assignment event
  // NtpServer.user pre-assignment event (before handler)
  srv.user = "root";
  // NtpServer.user assignment event
  // Transaction commit event
}
```

Table 3.13: Synchronous Events as Generalized Exception Mechanism

handlers may be defined by different parts of the same service, as well as remote JSpoon services. The synchronous configuration events therefore create a novel mechanism for adjusting program behavior over the management repository across process and host boundaries.

The JSpoon code fragment of table 3.13 performs two configuration operations on an `NtpServer` instance in an atomic transaction. The configuration events associated with the JSpoon actions, identify generalized exception handling points. Any JSpoon process may extend the behavior of a JSpoon thread executing this code fragment by associating synchronous event handlers. For example, the port assignment may be verified for conflicts with other services.

Monitoring Autonomic Behavior

Asynchronous events can be used to monitor the configuration and performance of autonomic services. All event subscriptions followed by a JSpoon block that is *not* preceded by the `synchronous` modifier are treated as asynchronous. All performance `monitor` notification are asynchronous, since they occur outside the context of a transaction.

```
Lease lease = monitor NtpServer for utilization > 0.9 over 30000 {
  // Deploy another NtpServer
  atomic {
    NetworkHost host = serverFarm.allocateHost();
    NtpServer ntpd = new NtpServer();
    ntpd.servedBy = host;

    Sequence<NtpClient> clients = event.object.ntpClients;
    host.ntpClient = clients.subsequence(0, clients.length / 2);
    event.object.clients = clients.subsequence(client.length / 2);
  }
}
```

Table 3.14: Asynchronous Performance Event-Triggered Configuration

Asynchronous configuration events are delivered to processes in transaction serial history order. For every transaction t, every event within t is delivered in the order in which it has occurred with in t, as previously discussed. Events occurring within an other transaction t_i are delivered in according to the order of the two transactions in the global serial history.

The example of table 3.14 shows a performance monitoring event handler triggered by high `NtpServer` utilization over 30 seconds. The handler executes a *new* configuration transaction which identifies an available network server host, and deploys an NTP service by creating an object and adding a relationship to that host. The clients of the over-utilized service are split between the old and the new servers. Note that this is a simplified example that does not take into account the source of the high utilization, or any global service goals.

3.3.3 Synchronizing States

JSpoon class configuration variables may be changed by other JSpoon processes. A change in the configuration variable will be reflected in program behavior at the next point at which this variable is read. For example, the `active` variable in the

program of table 3.1 is consulted every time a request is serviced. If the server is idle, then it will remain blocked on the socket `receive()` method, and will not terminate until a datagram has been received and processed. In order to increase responsiveness, the program may set a socket timeout to establish an upper bound on its delay to respond to a change in the `active` configuration.

Service Termination The JSpoon runtime environment monitors application access to configuration variables, and can detect if the program has failed to read a changed configuration value after a certain period. Based on its configuration, the environment can elect to restart the server. Because the Java VM does not support external thread termination (kill), JSpoon introduces an alternative termination mechanism. The `jspoon-checkpoint` keyword may be used to inform the JSpoon compiler that the thread can be safely terminated. Thread classes may implement the `jspoon.JSpoonShutdownHook` interface to effect additional thread termination cleanup. Alternatively, JSpoon threads may manually consult the static `jspoon.JSpoonThread.terminate` boolean variable in their operational loops, and effect thread termination when it is set to true.

Dynamic Configuration There are cases in which configuration is encapsulated in Java objects which are created once in the program's lifetime, typically during program initialization. Examples include network service port numbers, persistent storage directories, and others. Dynamic changes in configuration need to be propagated into the encapsulating object.

A manageable Java object is one whose state can be modified at runtime by invoking a set method. Manageable objects may be dynamically reconfigured in response to configuration updates. Programmers associate an event handler on the appropriate configuration property which invokes the set method on the encapsulating object.

A non-manageable object encapsulates its configuration state as an immutable attribute. Non-manageable objects must be re-instantiated in order to effect dynamic configuration. For example, main loop of the example from table 3.1 can be rewritten as:

```
while(active) {
  sock.receive(packet);
  if (port != sock.getLocalPort()) {
    // close & reopen socket
  }
}
```

It is possible to provide wrappers to non-manageable objects which effect the semantics of reconfiguration while maintaining object identity. A `MutableServerSocket` would encapsulate a `ServerSocket` adding a `setPort()` method. The `setPort()` method would queue all pending connections on the current socket, close that socket and then create a new `ServerSocket` instance. The `accept()` method would first return any old queued connections and then return connections from to the new server socket.

3.4 Extending the Management Schema

Knowledge plug-ins extend the capabilities of the basic JSpoon schema. For example, a simple constraint knowledge plug-in may add support for type range restrictions. A JSpoon autonomic element may use this plug-in to further restrict the values of port number configuration attribute to the range [1024..65535].

JSpoon knowledge plug-ins are enabled in the form of schema extensions to JSpoon classes. The schema of a JSpoon class is represented with a meta-object that is also a persistent JSpoon class of type `JSpoonClass`. `JSpoonClass` defines a to-sequence relationship to instances of the JSpoon class `JSpoonSchemaExtension`.

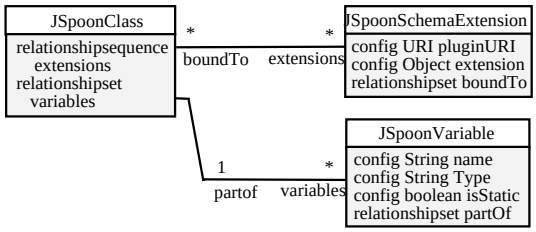

Figure 3.2: JSpoon Meta Schema

Schema extensions consist of a knowledge plugin URI and an opaque object. A schema extension may be bound to multiple instances of `JSpoonClass`. A UML class diagram showing parts of the JSpoon meta-schema classes is shown figure 3.2.

When the JSpoon runtime environment loads a JSpoon class, it attempts to retrieve the class meta-object from persistent storage. The schema extension relationship of the meta-object is queried and the runtime environment attempts to download any knowledge module proxy plug-ins that are not already installed based on the URI. This capability depends on the Java cross-platform and security features. Runtime changes to the meta-object schema extension relationship can also trigger the loading or unloading of knowledge plug-ins.

At load-time, plug-ins use the JSpoon runtime event subscription mechanism to request notification of object management events. Plug-in subscriptions provide synchronous notification of pending changes, enabling the knowledge module to abort invalid changes. Plug-ins receive two additional events triggered by the creation and closing of a transaction, or nested transaction. These events enable evaluation of postconditions.

The knowledge schema extension mechanism is very powerful, but may introduce cyclical computations. A cyclical computation may be triggered by the interaction of two knowledge plug-in modules that propagate changes. Because the periodicity of the cycle may be very large, it may be impossible to identify in a

reasonable time. Therefore, in absence of additional information on propagation paths, the JSpoon runtime system defines a maximum knowledge module iteration count.

3.4.1 SMARTS Event Correlation Plug-in

A key role of network and systems management in traditional non-autonomic networks is to identify and respond to abnormal conditions. This task is performed through monitoring of element instrumentation values for abnormal readings. An abnormal condition, such a network link failure, is typically associated with abnormal readings in multiple dependent elements. The challenge in such a situation is to identify the root cause of these cascading failures.

SMARTS InCharge[12] in an event-correlation system providing automated network root cause analysis. InCharge uses the MODEL[57] language to model network services using objects and relationships, and then express *problems* in terms of *symptom* events. Each problem, such as a "link down" is assigned an event signature. At run-time, network events are fed into a correlation engine which matches problem signatures to events using a codebook approach to determine the most likely root cause[58].

The SMARTS approach to root cause analysis automation depends on the existence of a network performance and problem propagation model. Currently, developing such models is a human-intensive process, because current service instrumentation does not expose internal or external dependency paths. Moreover, the correlation engine receives all network events without any mapping into threads of execution. Therefore, problem signatures must be made very robust to improve the correlation accuracy in noisy environments.

Alternatively, the SMARTS codebook engine can be expressed as a knowledge plugin over the JSpoon configuration model. This approach significantly re-

Figure 3.3: InCharge MODEL JSpoon Schema Extension Example

duces the modeling effort required. Performance events travel over JSpoon relationships defined by configuration properties. MODEL event and problem statements can be attached to JSpoon management models as schema extensions. For example, a "Failure" event is propagated from a network host, into all hosted services over the `serves` relationships. Figure 3.3 shows a simple JSpoon configuration model extended with MODEL event and propagation information. MODEL statements can be attached JSpoon schema extensions.

An InCharge JSpoon knowledge plugin would register for asynchronous events on management attributes appearing in MODEL expressions. The changes would be used as input to the codebook event correlation algorithm. Event correlation accuracy can be improved by reducing observation errors. JSpoon provides consistent views on all configuration changes, and can provide relative ordering of performance events.

3.4.2 Change Propagation Plug-in

The propagation of problems over relationships for the purpose of event correlation can be extended to configuration automation. The value of a service configuration property may be expressed as function over the value of other configuration properties. The effect of a change in a property, propagates over to other dependent properties. Using the JSpoon relationship construct, propagation functions

```
context InternetRadio: mtu :=
  servedBy.networkInterfaces
  ->select(i | not i.isLoopback)
  ->collect(i | i.mtu)->min())
```

Figure 3.4: Object Spreadsheet Language (OSL) Propagation Rule

may navigate cross-service dependencies. The following chapter is dedicated to a change propagation model and language called OSL for effecting automated service configuration.

A sample Object Spreadsheet Language (OSL) propagation rule is shown in Figure 3.4. The rule states that the Maximum Transfer Unit (MTU) of a UDP-based Internet radio streaming service should be set to the minimum MTU of the network interfaces of its execution environment. OSL statements can be attached as schema expressions to the class of their target attribute, *InternetRadio* in this example. OSL expression static analysis would generate a synchronous JSpoon event subscription, with the event handler recomputing the value of the `mtu` property as part of the event-generating transaction.

The OSL language may also be used to express declarative constraints over the JSpoon configuration model. A constraint knowledge plugin would subscribe to model events, and elect to abort transactions which violate constraints attached to the JSpoon schema as plug-in extensions.

3.4.3 Topology Discovery Plug-in

JSpoon service relationships express known service dependencies. For example, a web service depends on its network host, as well as the host's file system. In some cases, service relationships may be analyzed to extract wider system dependencies. For example, a JSpoon instrumented Ethernet interface will define a relationship to each other Ethernet interface in its cache. In switched environments, the con-

tents of the cache will be dependent on the host's communications patterns. For some applications, it is of interest to determine the link-layer broadcast domain. A link-layer topology discovery process could apply a simple transitive closure operation on the Ethernet interface relationships, in order to determine the broadcast domain. This domain could be represented as a new object `EthernetDomain`, with a relationship to each Ethernet interface.

An Ethernet topology discovery service could be implemented as a JSpoon knowledge plug-in. The plug-in would subscribe to changes in Ethernet interface object relationships, and incrementally evaluate the transitive closure to determine the broadcast connectivity. The JSpoon Ethernet interface class schema would be extended with a new relationship to a `EthernetDomain` object, and the event handler would establish such values on the instances. This idea can be extended to different connectivity domains, such as DNS resolution, Web proxying, and others.

3.5 JSpoon Compilation

JSpoon-enhanced classes are compiled into Java VM bytecode class files. The language was designed to support JSpoon-to-Java source-to-source compilation as an intermediate step. The JSpoon compiler generates two identically named Java classes for every JSpoon class. The element class contains both the management and non-management sections. The management class contains only the management variables and is used by remote elements. Both classes depend on the JSpoon runtime for view maintenance.

Management variable accesses by JSpoon programs are transformed into corresponding accessor methods. Left-hand-side (assignment) updates and right-hand-side reads are replaced by *set* and *get* method invocations. The actual management attributes are declared as private transient Java instance variables with mangled names. Configuration variable accessors invoke JSpoon runtime methods for lock-

ing and logging. Performance variable accessors invoke JSpoon event generation methods. A static block is also created to notify the runtime of a class loading event.

Atomic blocks are compiled into JSpoon runtime environment method invocations to create a transaction (potentially nested), commit the transaction at the end of the block, or abort if an exception has been thrown. The JSpoon compiler attempts to optimize lock acquisition through static code analysis to batch lock requests, and minimize lock upgrades. Batched lock requests are sorted by object ID to reduce the likelihood of deadlock. If the compiler can determine that an atomic block does not include access to configuration attributes, then no transaction is generated, only the events are batched.

The JSpoon compiler may optionally generate proxy objects for exporting the management schema to existing Java and XML-based persistence and management APIs and protocols. For example, the compiler may generate MBean objects conforming to the JMX[51] architecture to support interaction with JMX managers. The compiler may also generate an XML schema, and protocol proxy for XML management standards such as XMLCONF[59].

3.6 Related Work

3.6.1 Standard Management Platforms

Despite the central role of management in IT investment, the design of service management features is largely an ad-hoc process. Management features are typically introduced into new hardware and software services after the fact, based on customer feedback. Management requirements added a-posteriori to system design and implementation may violate design abstractions and lead to increased maintenance complexity. Newly exposed configuration properties create opportunities

for violation of unchecked implementation assumptions, posing risks to availability, performance and security.

The space of consistent service configurations is typically restricted based on constraints on internal as well as external operation. Today, the information required for consistent service configuration is buried in design documents, operations manuals, support forums, and code structures. Service management involves use of proprietary tools and protocols which have been developed to present low-level configuration and performance information to human operators. It is the responsibility of these expert operators to acquire the knowledge model needed to interpret the meaning of this information and effect configuration control.

Standard management protocols, such as SNMP[8] which were designed to reduce some of the complexity of managing multi-vendor systems have not effective in configuration management. Management Information Base (MIB)[60] standardization significantly trails new service development, and the schema contents are primarily focused on performance and status reporting. SNMP is particularly ill-suited to configuration management due to its lack of atomic operations, table row additions, and weak authentication and access controls. As a result, the standardized management infrastructure today is dedicated to performance and status gathering, while configuration occurs over proprietary management platforms using different management models, access protocols, and user interfaces.

Service discovery in such a fragmented environment becomes a challenging problem. Even within a single work-group, it is nearly impossible to automatically determine all available network elements and services. Probe-based approaches fail to compile accurate models because many relationships can only be discovered through analysis of element configuration. Expanding such analysis across domains, especially in the application layer, is unattainable with current management platforms. Although past services have had few inter-domain dependencies, emerging

services based on multi-tier architectures, and network virtualization technologies are creating new complex inter-domain dependencies. Today, such dependencies are typically discovered when an element failure propagates into other domains, possibly leading to cascading failures.

Retrofitting existing systems with autonomic functions is a very challenging problem due to these limitations in service instrumentation, synchronization, semantic modeling and discovery.

3.6.2 Network Modeling

The advantages of representing network element and service configuration using an object-relationship model have been established by several research projects and management standards [38, 11, 39, 15]. The categorization of management variables has been previously discussed in network and services management architecture research [61, 62, 18, 63]. This paper contributes to this research by specifying a language and mechanism for declaring restricted management variables as part of the service object class, and compiling them into a management object following matching accessor design patterns.

3.6.3 Management Automation

Previous work in automated service configuration[64, 65, 66, 32], fault root-cause analysis[12], and self-healing[67, 68] has depended on existing service instrumentation. Although these approaches have demonstrated practical automation capabilities, they require complex coordinated design and evolution of management systems and managed elements. For example, the management system vendor must be able to access the instrumentation of the managed element, construct a data model and a knowledge model to interpret its operational meaning and control its configuration in a manner consistent with operational procedures conceived by the

element vendor. This information is often not readily available and may change as the managed element continues to evolve. Therefore, there is a need to simplify the process through which element designers export instrumentation and modeling information concerning the element's operations management, while enabling multiple knowledge models to be seamlessly integrated and applied to the element to provide autonomic operations. This work provides an opportunity to greatly improve the efficacy of these results through a rich instrumentation layer, and an generic architecture for providing automation knowledge modules.

JSpoon complements existing Java-based management architectures such as JMX[51] and JavaBeans. Autonomic element programmer's benefit from the close integration between element and management code, while benefiting from the compiler-generated management system exports. Related research includes a system for automated management of EJB component interfaces[69]. The work presented proposes a more general instrumentation automation approach, at the cost of requiring programmer cooperation. Automated SNMP-based management instrumentation has been previously demonstrated in non-imperative languages such as the NetScript[70] data-flow language. This paper presents an architecture for automating instrumentation of Java, a general-purpose imperative language.

3.7 Conclusion

The JSpoon approach to autonomic management offers several substantive advantages over current alternatives. Management information is consolidated with element design and can be maintained through its life-cycle evolution. Instrumentation, data models, knowledge model and their bindings can be generated and managed through compiler support and static-time validation. Network events can be intercepted synchronously to constrain and extend the behavior of objects. Knowledge modules can be seamlessly incorporated with elements by vendors of

autonomic computing products, independently of the element vendors enabling synergistic evolution of products. instrumentation, data and knowledge models can be unified across multiple elements greatly simplifying the task of providing autonomic self-managing capabilities of large composite systems.

Chapter 4

Effecting Change Propagation

4.1 Introduction

This chapter introduces a novel approach to expressing autonomic behavior. Autonomic element configuration is computed as a spreadsheet function over a unified object-relationship model. Autonomic behavior is effected through propagation of changes across object relationships. The spreadsheet model requires that changes propagate in one direction, that is, there are no propagation cycles. An Object Spreadsheet Language (OSL) is introduced to express change propagation over object-relationship configuration models of JSpoon. OSL change rules are expressed as spreadsheet-style computations over the configuration model, as shown in figure 4.1. Autonomic element behavior results from the application of composable OSL libraries.

OSL rules express class model configuration attribute values as acyclic functions over the configuration attributes of related model classes. A change in a configuration attribute may trigger the evaluation of one or more rules in which it is used as a function parameter. Due to the restriction on acyclic propagation paths, there is no path of change which can propagate back to the triggering at-

tribute. Configuration operations that are dependent on performance attributes must first abstract these as more coarse-grain configuration attributes. For example, a configuration function that depends on server load, will be computed over a coarse-grained configuration attribute (e.g. low, high) whose update is triggered by a performance monitor.

OSL rules are of the form *field* `:=` *expression*, where *field* is an management attribute or relationship, and *expression* is a side-effect free function over the object management attributes, and relationships. Rule evaluation is triggered by synchronous modeler event notifications. Three incremental (OSL) extensions of an are introduced, as illustrated in figure 4.2. The basic OSL_0 language provides assignment, arithmetic operations, and simple navigation of relationships. The next language $OSL_{0.5}$, which is a superset of OSL_0, adds a conditional operator. Finally, the highest language, OSL_1 adds iteration and existential operators which can be used to expressing first order queries.

The chapter presents *algorithms for static analysis* of OSL rule-sets to assure that autonomic elements maintain configuration safety within the dynamic environments in which they are installed. Static analysis will be typically performed at design time to assure safety. All OSL rule sets can be accurately analyzed for termination. Under certain rare conditions, analysis of $OSL_{0.5}$ and OSL_1 rule sets may result in false detection of non-termination conditions. The results of static analysis are used by algorithms for efficient computation of change rules.

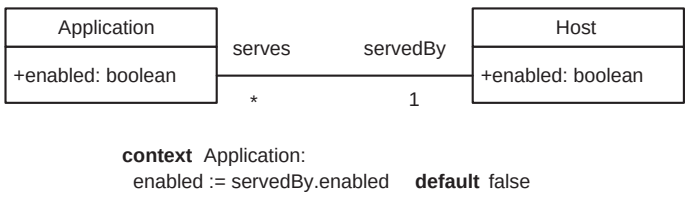

Figure 4.1: Object-Relationship Spreadsheet Model

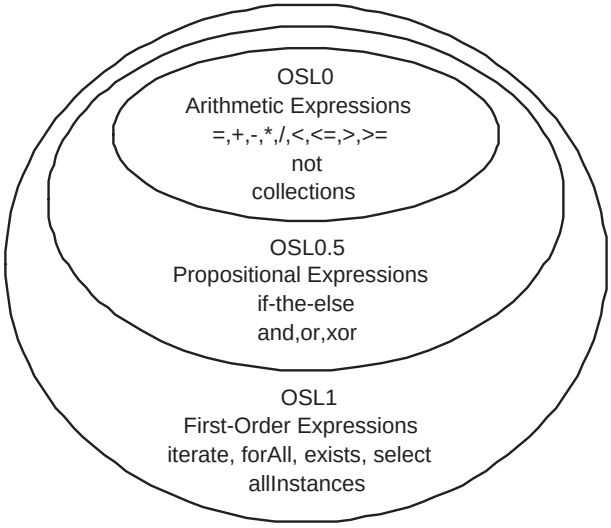

Figure 4.2: Object Spreadsheet Language Extensions

A declarative subset of OSL called the Object Policy Language (OPL) is used to express and enforce configuration *policy constraints*. The ability to propagate changes across multiple autonomic elements introduces the risk of system failure due to programming errors. Any change propagation system must guard against arbitrary changes by enabling the specification of constraints on the configuration of individual, as well as related services. Policy constraints are a tool for protecting systems from failures due to invalid propagation of changes. Their evaluation is triggered by synchronous modeler event notifications.

Scalability of change propagation poses several challenges. Rule evaluation must be performed incrementally to reduce processing overhead. Rule analysis must be performed against the static schema model as its instantiation will typically be larger by several orders of magnitude. Finally, the scope of propagation must be analyzed and restricted to prevent failure propagation across domains, and assure rule scalability. This thesis chapter introduces novel contributions to these challenges in the form of algorithms for static rule analysis over the object-relationship

schema, incremental change propagation evaluation algorithms, and a hierarchical domain-based architecture.

4.1.1 Change Propagation Challenges

Presently, individual system behavior is automated using general purpose scripting and programming languages such as Bash, Perl, Python, C/C++, Java, and Ada. Although such languages have been successfully applied to individual large automation projects, such as anti-lock brake systems, missile control systems, and simulations, they have had limited success as system integration languages. Besides the concurrency, recoverability, and behavior instrumentation challenges identified in the previous two chapters, automation *system interaction* presents a major technical challenge.

Current configuration automation programs are typically tailored to specific features, such as propagation of DHCP leases to DNS databases, or the propagation of a new employee record into the the building security access database. These automation features may interact, causing inconsistencies and re-configuration cycles that may span administrative domains and manifest with varying periodicity. In general, it is impossible to determine apriori the interaction of programs written in Turing-complete languages. As a result, deployment of automation features is severely restricted in current environments.

Current approaches to management automation in the form of scripts, or embedded self-management functions have proven inadequate because the do not support mechanisms for composition, analysis, and scalability. A management automation system must support the following capabilities:

- Effective mechanisms for coordinated automated changes of multiple interdependent elements. Current management architectures do not support transactional change configuration semantics.

- Composition of configuration scripts from different elements (and respective scalability). Current automation programs depend on raw configuration input and output, and cannot be directly composed. Composition through configuration side-effects can lead to non-termination or non-deterministic behavior.

- Static analysis of configuration scripts to ascertain that the changes they introduce in all instances of networks are safe. Currently, automation programs operate over raw repositories with full access permissions. It is not possible, in general to determine the exact effects of an automation program. Therere, programmed changes cannot be rolled back in the case of failure.

- Enforce policy constraints support control of propagation of changes across domains. Current programs cannot be easily monitored since they operated using multiple protocols over raw local element repositories. Local repositories do not contain the dependency information required to verify policy.

4.1.2 Management Schema Example

The change propagation and policy languages presented in this chapter will be illustrated using a common management schema example. The sample management schema is illustrated in figure 4.3 using the notation of UML[17]. It was was chosen to include all the core JSpoon management schema features. It should be noted that the example was selected with the goal of providing a minimal schema illustrating available features, and is not intended to represent an actual unified configuration model.

The sample schema declares an `Application` class containing two boolean management attributes. The `enabled` attribute controls the operation of the application, while the `active` monitors the status of the application. A subclass of the `Application` class is used to define an Internet radio service with an attribute con-

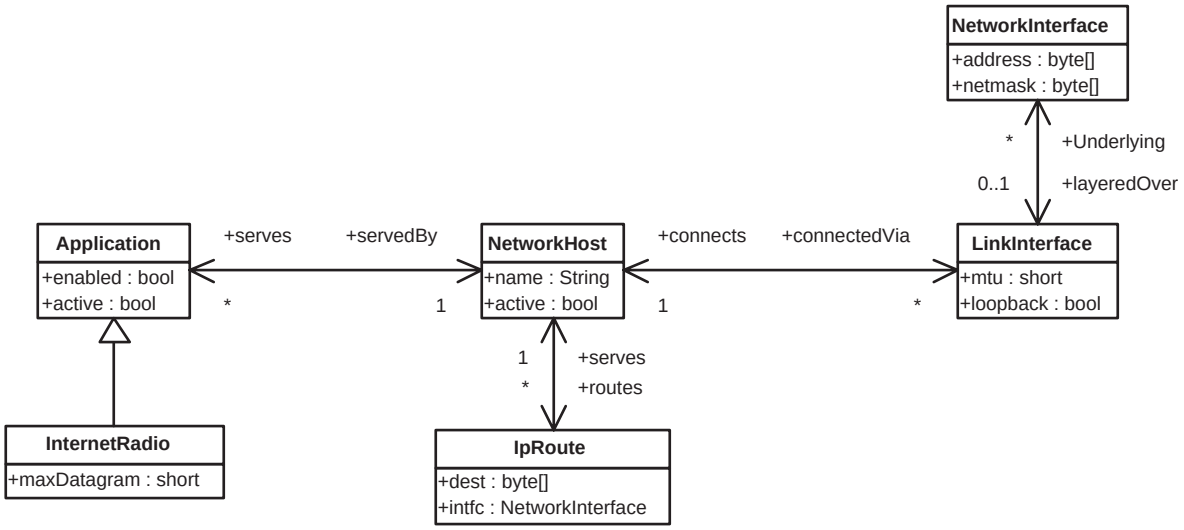

Figure 4.3: Example Management Schema

trolling the maximum size of the broadcast datagrams. The `Application` class is related in a many-to-one relationship to a `NetworkHost` class, which acts as its host. In a similar manner, the `NetworkHost` class is related to a `LinkInterface` class that encapsulates a link-layer network interface. Multiple network interfaces can be layered over a link interface as `NetworkInterface` instances. Finally, the network host has an associated routing table which is modeled as a to-many relationship to `IpRoute` object instances.

The chapter is structured as follows. The next section introduces three incremental extensions of a language for expressing changes over a JSpoon object-relationship model. The following section introduces a formal model of spreadsheet rule evaluation, and desired properties, and presents efficient algorithms for establishing these properties. Policy constraints are then introduced in the next section, as declarative OSL expressions. Hierarchical domain scalability is the subject of the next section. The chapter concludes with discussion of previous related work.

4.2 Expressing Object-Relationship Change Propagation

In the object-relationship model, configuration and performance properties are expressed as typed class attributes, while dependencies are expressed as binary relationships between classes. A change propagation language operating over an object-relationship model must support navigation of the schema features, and define the semantics of these operations.

For example, an UDP-based Internet radio application depends on the ability of the network to transmit datagrams to its listeners. If the size of the datagrams exceeds the network's Maximum Transfer Unit (MTU), they may be fragmented or dropped. Therefore, it is useful to define a rule that propagates the network's MTU to the configuration of the service's datagram size. All the required information is available in the unified object-relationship model. While it is possible to use a general purpose programming language, such as Java, to access and manipulate the model, such programs cannot be analyzed for interaction with other automation programs.

This chapter introduces three incremental extensions of a Spreadsheet Object Language (OSL) for expressing such propagation of changes. As will be discussed in in the next section, each increase in expressive power effects the analysis and evaluation of rules expressed in each language. All languages share the fact that they are strongly typed, against the JSpoon type model, and define spreadsheet rules which propagate the result of evaluating an expression into a class attribute or relationship.

The main tradeoff between the spreadsheet change propagation model and Turing-complete scripting is between analysis and expressive power. The OSL languages have a restricted set of operators which admit static analysis for termination,

independent of model instantiation. The restriction on acyclic rule evaluation restricts OSL programs from expressing policies requiring transitive closure. As will be demonstrated in the next chapter on applications, it is still possible to express a wide range of policies in configuration automation, security, and autonomic networks.

The syntax of OSL is derived from the syntax of the Object Constraint Language (OCL)[71], which itself is derived from the syntax of SmallTalk.

OSL Types

OSL is a strongly typed expression language over the JSpoon type system. OSL_0 expressions may access and manipulate JSpoon configuration attributes, and relationships. OSL supports static type-checking of expressions[72]. The following JSpoon types may be manipulated:

- *Primitive* types: the standard Java language primitive types (`boolean`, `byte`, `char`, `short`, `int`, `long`, `float`, `double`. In addition, OSL defines a `string` primitive type which represents an immutable Unicode string without support for the Java `String` objects operations. The three languages provide increasingly powerful operations on these primitive types.

- Java *immutable* classes: Java library classes representing immutable objects, whose state cannot be modified past construction time, may be accessed. Such objects can only be compared for equality and no methods may be invoked on them.

- JSpoon objects: JSpoon objects may be used as types of configuration attributes or as endpoints in a relationship. Only the management section attributes and relationships are visible. No Java methods may be invoked on such objects.

```
on (external change to field f of object o of class c on transaction t) do
    m ← true
    while(m) do
        m ← false
        for (each rule r) do
            for (each instance o' of of rule's target class c) do
                if o'.t ≠ e(o') then
                    o'.f ← e(o')
                    m ← true
```

Table 4.1: OSL Evaluation Semantics

Evaluation Semantics

Every OSL propagation rule is of the form "$c.t$:= $e(o)$", where t is an attribute or relationship of class c, and f is a side-effect free expression over object o which is an instance of c. The operations supported in the expression component are defined by the successive OSL languages. It is safe to evaluate the expression $e(o)$ any number of times, since its evaluation cannot effect the state of the model. The *only* side-effect possible in OSL is through the assignment operation ":=". In this sense, the spreadsheet model is a hybrid of the declarative and imperative programming styles[73].

Only one rule for each class attribute or relationship may be declared in OSL. This restriction assures that there rule evaluation order does not effect the final propagation state. For example, the MTU of a host's link interfaces can be computed by navigating the network membership relationship to retrieve the gateway MTU.

Unlike traditional Event-Condition-Action (ECA) systems[41], the semantics of OSL evaluation state that *every* formula is re-evaluated on *every* instance after *every* model change The propagation algorithm, shown in table 4.1, is invoked whenever a change occurs on the instantiation of the object-relationship model.

There are four types of possible changes that can be performed on an instance of the model:

- *attribute modified*: the value must be propagated by computing every formula in which it is used,
- *relationship modified*: new propagation paths may be created, and formulas depending on the state of the relationship will have to be evaluated,
- *object created*: the value of each attribute and relation that is the target of a rule needs to be evaluated. A change in one field may result in changes to the same, or other objects.
- *object removed*: the object is removed from any relationships in which it participates, and the formulas operating over these relationships are evaluated.

If cyclical rule declarations, such as "$c.x := y$" and "$c.y := x$" are permitted, then the loop may never terminate. It will be shown that the OSL spreadsheet languages can be analyzed statically for termination, and that the evaluation of an acyclic set of OSL rules will always terminate. An incremental algorithm will be demonstrated in which a formula is evaluated incrementally when its expression components change.

4.2.1 OSL_0 Arithmetic Expression Language

The simplest form of change propagation language is one allowing the propagation of values over the object-relationship model without conditional statements and quantification.

Operation	Notation	Equivalent	Result Type
equals	`a = b`		boolean
negation	`not a`	`a = false`	boolean
not equals	`a <> b`	`not (a = b)`	boolean

Table 4.2: OSL_0 Boolean Operations

Operation	Notation	Equivalent	Result Type
equals	`a = b`		boolean
addition	`a + b`		number
subtraction	`a - b`		number
multiplication	`a * b`		number
division	`a / b`		number
less	`a < b`		boolean
less or equal	`a <= b`		boolean
more	`a > b`	`not (a <= b)`	boolean
more or equal	`a >= b`	`not (a < b)`	boolean
not equals	`a <> b`	`not (a = b)`	boolean
negation	`- b`	`0 - b`	number

Table 4.3: OSL_0 Arithmetic Operations

Operations

OSL_0 restricts the types of operations that may be performed on values. Primitive boolean types can be compared for equality. Negation and not-equals follow from boolean equality as shown in table 4.2. Binary boolean operators, such as "and", "or", and "xor" are not available in OSL_0.

Primitive arithmetic types may be compared for equality and inequality, added, subtracted, divided and multiplied. Table 4.3 shows how negation and not-equals follow from these operations. Division introduces the possibility of undefined results. The `default` operator which will be introduced later in this section may be used as an exception handling mechanism.

Java objects may only be compared for reference equality. No methods may be invoked on such objects. JSpoon object configuration attributes and relationships can be accessed, but none of their Java methods may be invoked.

Attribute Assignment

A language for automating change propagation in an object-relationship model must, at a minimum, support the setting of object attributes. The expression below states that the `mtu` configuration attribute of all instances of class `InternetRadio` is assigned the constant value `508`. The `context` statement defines the class scope.

```
RULE 1 : context InternetRadio: mtu := 508
```

Class attributes may also be assigned values from other class attributes of the same instance. For example, the `active` performance attribute is assigned the current value of the `enabled` configuration attribute.

```
RULE 2 : context Application: active := enabled
```

The semantics of assignment in OSL are different from that of assignment in imperative languages, such as Java. An OSL assignment expression is a constraint on the value of an attribute. It states that whenever the class attribute is accessed, its value will be equal to the result of evaluating the right-hand-side expression. Note that this specification allows for lazy evaluation of OSL rules.

Only a *single* assignment statement may be defined for each object class attribute. The OSL runtime system will not allow multiple assignments to the same attribute.

Navigating Relationships

The source of configuration values may include other objects that are reachable via relationships. In the example below, the `active` property is assigned the value of the `active` property of the `NetworkHost` object related to the `Application` object through the to-one `servedBy` relation. The dot "." operation is used to access the `active` property of the object in the `servedBy` relation.

```
RULE 3 : context Application : active := servedBy.active
```

Because `servedBy` is a to-one relationship, the result of dereferencing it will be either a single instance of `NetworkHost`, or `null`. The previous rule will therefore be undefined when applied to an `Application` instance which is not related to a `NetworkHost` instance.

OSL requires that assignment expressions must always be defined. An exception handling mechanism is provided to handle null relationship dereferencing events. In the example below, the `active` property is set as before, but if a null relationship is navigated, then the default value of `false` is assigned. All rules navigating to-one relationships must provide such handlers.

```
RULE 4 : context Application :
         active := servedBy.active default false
```

Collection Types

Navigation of to-many relationships requires the introduction of collection types. The management schema language supports the declaration of two types of to-many relationships: to-set and to-sequence. The result of navigating a to-set or a to-sequence relationship will be a set, or a list respectively. Collections are immutable and are parameterized by the type of the relationship endpoint. In the OSL_0

Operation	*Result*	*Description*
`size()`	`int`	number of elements
`isEmpty()`	`boolean`	true if collection is empty (`size = 0`)
`toArray()`	*Type*`[]`	returns elements as array of *Type*
`collect(name)`	Collection	collects an attribute or relation from each element

Table 4.4: OSL_0 Collection Operations

assignment language, only three collection operations are supported as listed in table 4.4.

The next expression shows an example rule for setting the `active` property of the `NetworkHost` class to be true if the host is connected to one or more link interfaces. The `connectedVia` relationship is a to-set relationship, and therefore evaluates to a value of type `Set<LinkInterface>`. In order to improve readability, collection operations use the arrow "`->`" operator, instead of the dot "`.`" operator.

```
RULE 5 : context NetworkHost:
         active := not connectedVia->isEmpty()
```

Collection types support the grouping of attributes or relations. For example, if the `NetworkHost` class had a `byte[][] addresses` property, we could express a propagation rule that navigated the `connectedVia` relation to obtain a set of `LinkInterface` objects, and then navigate the `layeredOver` to-one relation to collect the network interface address of each link interface.

```
RULE 6 : context NetworkHost: addresses := connectedVia
         ->collect(underlying)->collect(address)->toArray()
```

In order to improve readability, the dot "." operator is overloaded to act as a *collect* operation on sets (OCL feature). The example below is equivalent to the previous one:

```
RULE 7 : context NetworkHost: addresses :=
    connectedVia.underlying.address->toArray()
```

It should be noted that the above examples navigate the to-one `underlying` relationship which may be undefined. Therefore, the expression must catch that error as shown below:

```
RULE 8 : context NetworkHost: addresses := connectedVia
            .underlying->collect(address->toArray() default null)
```

Relationship Assignment

The assignment operator can be used to specify the values of relationships as well as attributes. Only a single assignment may be defined for each object class relationship. Moreover, only one end-point of the relationship may be associated with an assignment rule. The OSL runtime system will not allow multiple assignments to the same relationship, or concurrent assignments to both endpoints.

To-many relationships may be assigned from arrays or collections of objects of the relationship target type. If an ordered to-many relationship (list) is assigned from a set, then the order of the elements will be arbitrary. A rule may construct a set or sequence explicitly using the special collection constructor:

```
RULE 9 : context NetworkHost:
  routes := Set<IpRoute>(IpRoute("0.0.0.0/0", "eth0"),
                         IpRoute("127.0.0.0/8", "lo0"))
```

The set and sequence collection constructors take variable-length arguments, which are used to build the set or sequence. The order of the arguments is important in sequence declarations.

Object Creation

Tabular information is frequently encountered in network configuration management. Examples of common tabular configuration data include route tables, firewall rules, and name-address tables. In the typical mapping of such information to the object-relationship model, a class is defined containing the table column attributes. Instances of that class represent table rows. These may be stored in a Java array, or using an ordered, or unordered to-many relationship.

OSL rules may create objects to populate arrays or relationships. For example, consider a rule computing the route table of a network host. The `NetworkHost` object is related via the `connectedVia` relation to zero or more link interfaces. Each `LinkInteface` is used to support zero or more network interfaces. The OSL rule below sets the network host routes relationship to the set of `IpRoute` objects created for each network interface. Instances of `IpRoute` are created by invoking the class constructor.

```
Rule 10 : context NetworkHost: routes :=
  connectedVia.underlying->collect(IpRoute(netmask, this))
```

Object creation is an additional mechanism for creating side-effects. The objects created may trigger additional rules, and require life-cycle management. The semantics of OSL rule object creation are that during every rule evaluation, all previously created objects are deleted, and new ones are created. Therefore, a rule-created object cannot exist outside the relationship in which it was defined. In practice, implementation will optimize by recycling objects, as long as this practice does not violate the aforementioned semantics.

Additionally, rule-created objects are restricted as follows:

- Objects are created an object constructor of the form: *class-name* "`(`" *argument* ("`,`" *argument*)* "`)`"

Expression	Description
set += object	add object to set
set += set2	merge sets
set := { object } ∪ set2	result

Table 4.5: Relationship Set Union Semantics

- All object attributes and relationships must be specified at construction time,
- Object created by rules are not persisted,
- Object identity may change and should not be depended to remain stable.

Distributing Relationship-Set Assignments

The semantics of relationship assignment state that there may be only one assignment per class relationship, and only one side of the relationship may be assigned to. It some cases, it may be more natural to express the assignment of an unordered to-many (set) relationship as a union of sets. For example, entries to the route table may be defined by the network interface object as shown below. This definition is equivalent to the example from the object creation discussion.

```
RULE 11 : context NetworkInterface:
  layeredOver.connects.routes += IpRoute(address, netmask)
```

The plus-equals "**+=**" operator is defined as a union operation on relationship sets. It is permissible to define any number of plus-equals rules on the same to-many set relationship, as long as no assignment rule has been defined. As shown in table 4.5, the multiple plus-equals rules are converted into a single assignment rule containing a union of all the right-hand-side expressions.

Let Environment

The `let` statement may be used to define local constants and functions within an OSL expression (OCL feature). The let statement syntax is:

"`let`" *variable* { "`(`" *parameter-list* "`)`" } { "`:`" *type* } "`=`" *expression* "`in`" *expression*

For example, the example below defines the local constant `DEFAULT_MTU` for use within the computation.

```
Rule 12 : context NetworkInterface:
  mtu := let DEFAULT_MTU = 508 in DEFAULT_MTU * 2
```

A local function may also be declared in the let environment as shown in the next example:

```
Rule 13 : context LinkInterface:
  loopback := let isLoopback(IpAddress addr) : boolean =
                addr = IpAddress("127.0.0.1")
              in
                isLoopback(underlying.address)
```

Management Functions

OSL expressions may be parameterized and declared as management functions attached to a context object. This mechanism allows OSL programmers to extend the methods of a JSpoon object at runtime. Management functions may be overloaded with multiple dispatch, but may not override or overload Java methods.

```
Rule 14 : context IpRoute: defun isLocal() : boolean =
  intfc.layeredOver.loopback default false
```

Embedding Management Functions in JSpoon

OSL expressions can be statically associated with JSpoon class definitions. Assignment rules are added as JavaDoc[74] "`@jspoon OSL :=`" tag associated with the JSpoon attribute declaration. In the example below, the `active` attribute of the `Application` class is declared as a JSpoon configuration attribute. The JavaDoc JSpoon tag specifies that the value of the attribute is computed by the expression provided. At compile time, the JSpoon compiler will invoke the OSL plugin to parse the expression, and extend the `Application` class JSpoon meta-schema.

```
RULE 15 public class Application {
  /** @jspoon OSL := servedBy.active default false */
  config boolean active;
}
```

Similarly, management functions may be declared as JavaDoc comments placed in the class context as shown below.

```
RULE 16 public class Application {
  /** @jspoon OSL defun isUp() : boolean = ... */
}
```

4.2.2 $OSL_{0.5}$ Propositional Expression Language

Conditional Operation

$OSL_{0.5}$ extends OSL_0 with an `if-then-else` operation. Since there are no side effects in OSL expressions, the semantics are different from those of imperative languages. The `if-then-else` operation evaluates the `if` condition, and if it is true, it returns the result of evaluating the `then` expression, otherwise it returns the result of evaluating the `else` expression.

Operation	Notation	Equivalent	Result Type
and	`a and b`	**if** `a` **then** `b` **else** `false`	boolean
or	`a or b`	**if** `a` **then** `true` **else** `b`	boolean
xor	`a xor b`	**if** `a` **then not** `b` **else** `b`	boolean

Table 4.6: $OSL_{0.5}$ Boolean Operations

The $OSL_{0.5}$ example below configures the maximum transfer unit (MTU) of a link interface based on the type of its interface. If the `loopback` `LinkInterface` attribute is true, then the `mtu` is set to 16436, otherwise it is set to 1500.

```
RULE 17 : context LinkInterface mtu :=
  if loopback then 16436 else 1500
```

Binary Boolean Operations The OSL_0 language is restricted in the types of operations on booleans. For example, the lack of binary boolean operators, means that rules such "an application is active if it is enabled and its host server is active" cannot be expressed. The $OSL_{0.5}$ language with boolean binary operations which are short-hands for the application of the `if-the-else` operation as shown in table 4.6. The aforementioned rule can thus be expressed as shown in the next example:

```
RULE 18 : context Application: active :=
  enabled and (servedBy.active default false)
```

Switch Table Operation When defining management change propagation rules it is common to encounter expressions which associate values based on a table. For example, the MTU of a link interface is typically determined by its type. Although it is possible to express such tabular rules using nested `if-then-else` expressions,

"switch" "(" *int-attribute* ")" { ":" *type* }
"case" (*int-value*$_1$ ":" *expression*$_1$ ",")*
"default" ":" *expression*$_n$

if (*int-attribute* = *int-value*$_1$) then *expression*$_1$
else if (*int-attribute* = *int-value*$_2$) then *expression*$_2$
else *expression*$_n$

Table 4.7: Switch Operation Syntaxt & Semantics

the $OSL_{0.5}$ `switch` operation provides improved readability. The syntax of the operation and its semantics in terms of `if-the-else` are shown in table 4.7.

Using the $OSL_{0.5}$ `switch` operation, the MTU of the link interface could be set as shown in the next example:

```
RULE 19 : context LinkInterface mtu :=
  switch(type) : int
    case LOOPBACK: 16436,
    case ETHERNET: 1500,
    case PPP:       576,
    default: 1500
```

4.2.3 OSL_1 First-Order Expression Language

The $OSL_{0.5}$ language cannot express iteration over collection types, or instances of a JSpoon class. Rules such as "a network host is connected to the network if it has a network address configured over a non-loopback interface" cannot be expressed in $OSL_{0.5}$ because one cannot iterate over the `connectedVia` relationship and select only the link interfaces whose `loopback` attribute is true.

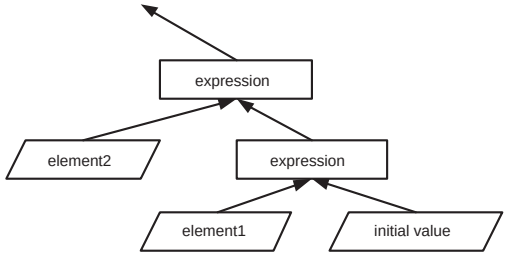

Figure 4.4: Iterate Operation Flowchart

Iteration

The OSL_1 language adds a first-order collection operation iteration operation. The `iterate` operation iterates over each element of the collection, evaluating an expression over the element and an input value. On the first iteration, the input value is provided by as an expression to the operation, while in the subsequent iterations it is the result of the previous expression computation, as shown in figure 4.4. The expression evaluates to the result of the last iteration.

Selection

For example, consider a management function which counts the number of network interfaces configured on a host. The example below defines a management function that iterates over all host link interfaces, and counts the number of network interfaces layered over them. The counting is achieved by starting initially at zero, and adding the number of network interfaces as an accumulator.

```
RULE 20 : context NetworkHost :
  defun networkInterfaces() : int =
    connectedVia->iterate( l : LinkInterface; count : int = 0
                           | count + l.underlying->size())
```

Operation	*Result*	*Description*

`select(`*bool-expr*`)`	`Collection`	the elements satisfying the expression.
		≡ `iterate(s : Set<`*Type*`> |`
		`if (`*expression*`) then Set<`*Type*`>(s, this)`
		`else s)`

`reject(`*bool-expr*`)`	`Collection`	the elements *not* satisfying the expression. ≡ `select(not` *bool-expr*`)`

Table 4.8: OSL_1 Collection Selection Operations

Operation	*Op.*	*Result*	*Description*
`forAll(`*bool-expr*`)`	`and`	`boolean`	true if expression is true for *all*. ≡ `iterate(a = true | a and` *bool-expr*`)`
`exists(`*bool-expr*`)`	`or`	`boolean`	true if expression is true for *one or more*. ≡ `iterate(e = false | e or` *bool-expr*`)`
`sum(`*num-expr*`)`	`+`	*number*	sums the expression over *all*. ≡ `iterate(c = 0 | c +` *int-expr*`)`
`min(`*num-expr*`)`	`<`	*number*	returns the min. value in the collection. ≡ `iterate(min = Integer.MAX |` `let v =`*int-expr* `in if (v < min) then v else min)`

Table 4.9: OSL_1 Collection Accumulator Operations

The *iterate* operation may be used to select objects from collections. Because selection is a common operation, OSL_1 provides explicit selection operations as listed in table 4.8. In the next example, the `select` operation filters the elements of a set by evaluating an expression

```
RULE 21 : context NetworkHost :
  defun isConnected() : boolean = connectedVia
  ->select( (not loopback) and
            (not underlying->isEmpty()) )->size() <> 0
```

4.3 Expressing Policy Constraints

Policy constraints set restrictions on admissible configurations. Policies can be general or specific to a given domain. Domain-specific policies cannot be embedded in the JSpoon program code since they must costomized to domain requirements. Policies gate the applications of change rules. If a policy forbids a derived change, it will cause rejection of the source transaction.

Policies are used to assure that updated configurations meet operational policies. For example, a server should not serve more than a number of simultaneous clients, or an application should not admit more than so many instances allowed by a licensing policy. Policies may also be used to prevent vulnerabilities to failures. For example, an attribute may be changed ot most every 5 minutes.

Policy constraints are declarative expressions over the object-relationship model evaluating to a boolean value. Constraints are used to validate changes in the model instantiation. The navigational requirements for expressing policy constraints over an object-relationship model are very similar to those of expressing propagation rules. However, unlike propagation rules, constraints do not effect the model, and therefore there is no possibility of mutual interaction. There are two basic types of constraints:

1. *invariants*: expressions that must evaluate to *true* after every change has been applied and all applicable change propagation rules have been applied.

2. *postconditions*: expressions that must evaluate to true at the end of a transaction.

The semantics of constraint evaluation are expressed in table 4.10. In the autonomic management architecture, all model access is transactional. External changes to the model are all associated with a transaction t. Invariant evaluation is triggered by a *every* external change, and is performed after the all changes

on (each external change to field f of object o of class c on transaction t) do
    propagate changes (table 4.1)
    for (each invariant constraint a) do:
        if (not verify(a)) abort(t)

on (transaction vote) do
    for (each postcondition constraint p) do:
        if (not verify(p)) abort(t)

Table 4.10: Constraint Evaluation Semantics

triggered have been propagated. If an invariant is found violated, then the transaction is aborted. Postconditions are evaluated when the transaction initiator has requested a transaction commit, and the transaction enters its voting state. If any postcondition has been violated by the transaction model updates, external as well as propagated changes, then the transaction is aborted.

As was the case with propagation rules, an implementation is allowed to optimize the verification of constraints based on triggering dependencies, as long as the above semantics are satisfied.

The change propagation language hierarchy was necessitated by the complexity of analyzing cyclical dependencies between rules. Policy constraints do not require such analysis, therefore OSL_1 can be used as a basis for the Object Policy Language OPL. The general form of OPLconstraints are shown below. The constraint type may be either `inv`, or `post`, and the boolean expression may be any OSL_1 expression evaluating to a boolean value.

`context` *class-name* : *constraint-type* : *boolean-expression*

4.3.1 OPL Implies Operator

The "e_1 `implies` e_2" operator may be used to assert the truth of y if x is true. It is equivalent to the $\text{OSL}_{0.5}$ statement "`if` (e_1) `then` e_2 `else true`", where e_1, e_2 evaluate to a boolean value.

4.3.2 OPL Invariants

An OPL invariant constraint is declared as an expression specifying the `context` of the expression evaluation, the constraint type `inv`, and an expression evaluating to a boolean value. An example invariant is shown below. The invariant states that the link interfaces of a network host should only be enabled if the host's internal firewall has been enabled. The invariant restricts the order in which changes may be performed or propagated in the same transaction. Invariants typically prevent race condition in application of the effects of a transaction.

```
CONSTRAINT 1 : context NetworkHost : inv :
  connectedVia->forAll((enabled = true) implies firewall.enabled)
```

Invariants can conflict with rule-based propagation of changes. It is the responsibility of the rule programmers to assure that their rules do not violate invariants irrespective of the order applied. For example, a rule for configuring the `enabled` attribute of a `LinkInterface` would need to be encoded as shown in the next example, to guard against violation of the constraint:

```
RULE 22 : context LinkInterface : enabled :=
      (connects.firewall.enabled default false) and (linkSignal)
```

4.3.3 OPL Postconditions

Postconditions are used to express constraints on the result of a transaction consisting of external changes as well as rule-based propagations. Unlike invariants, postconditions may be violated in the course of a transaction, but must be satisfied in the final transaction state.

A postcondition constraint example is shown below. The constraint states that in the context of an `Application` object instance, if the `enabled` attribute is set to `true`, then the application must be related to a `NetworkHost` instance through the `servedBy` relation.

```
CONSTRAINT 2 : context Application :
  post: enabled implies not servedBy <> null
```

4.4 Static OSL Analysis

The semantics of the spreadsheet object-relationship model do not allow for cycles in rule evaluation. There are two basic approaches to rule cycle (termination) checking. Static rule analysis is used to determine if there is *any* possible model for which the rules will evaluate cyclically. The other approach is to perform cycle checking at runtime by tracking the propagation of changes over the model instantiation.

For the purposes of automation, static analysis is the preferable option since autonomic systems are expected to operate without direct human control. The discovery of cycles at runtime would require external intervention in order to analyze the the nature of the cycle, and break its occurrence by removing or modifying one or more rules. For example, an autonomic service may configure its execution priority to be higher than all other services. Introducing another service with the same policy will lead to a very long "bidding" war between the two service. Runtime

detection of such a cycle will require may only be resolved through the removal of one of the services. Had that cycle been detected at design-time, an appropriate policy could be established.

Given a Turing-complete change propagation language, such as an Event-Condition-Action system, it is not possible, in general, to statically analyze a set of rules for cyclical evaluation. This follows from the fact that it is not even possible to analyze the rules for termination (halting problem). The OSL languages introduced in the previous section, however, are not Turing-complete and therefore it is possible to perform static analysis in order to determine rule dependencies.

This section introduces a formal spreadsheet model and presents a graph-based approach to rule analysis and incremental evaluation.

4.4.1 Spreadsheet Model

The semantics of spreadsheet languages are a mixture of imperative and functional semantics. A spreadsheet rule is typically of the form:

$$x \leftarrow expr(x_1, x_2, ..., x_n)$$

where *expr* is an expression without side-effects over the variables $v_2, v_3, ..., v_n$. Changes to the state of variables in the spreadsheet model can only be performed using the assignment operator ($\leftarrow$).

Consider a program language L consisting of rules in the form

$$y \leftarrow f(x_1, x_2, ..., x_n)$$

where $y, x_1, x_2, ..., x_n$ are variables, and f is a function over the variables $x_1, x_2, ..., x_n$.

Definition 1 (Trigger) For a rule s in L, let $Trigger(s)$ be the set of variables appearing as triggers in the right-hand-side expression $(x_1, x_2, ..., x_n)$.

DEFINITION 2 (TARGET) Let $Target(s) \in V$ be the right-hand-side variable of the asignment ($\leftarrow$) statement (y).

Note that the above definitions do not preclude the existence of a statement $s \in L$ such that $Target(s) \in Trigger(s)$, but require that for all $s \in L, Trigger(s)$ is a set.

DEFINITION 3 (CHAIN) A *chain* is a sequence of rules $s_1, s_2, ..., s_n$ in L such that $n > 0$ and $Target(s_1) \in Trigger(s_2)$, and $Target(s_2) \in Trigger(s_3)$, ..., and $Target(s_{n-1}) \in Trigger(s_n)$.

The size of a chain is the number of rules in the sequence. A chain of size 1 is called a trivial chain. Two chains $s_1, s_2, ..., s_n$, and $t_1, t_2, ..., t_m$ are equal, if and only if, $n = m$ and $\forall 1 \leq i \leq n : s_i = t_i$.

DEFINITION 4 (CYCLE) A change propagation program is said to contain a *cycle* if it contains a chain s_1, s_2, ..., s_n, such that $Target(s_n) \in Trigger(s_1)$.

Note that it is possible to have a looping trivial chain if it contains a rule s for which $Target(s) \in Trigger(s)$.

DEFINITION 5 (AMBIGUITY) A change propagation program is said to have *ambiguity* if it contains two non-equal chains $s_1, s_2, ...s_n$, and $t_1, t_2, ...t_m$, such that $Trigger(s_1) \cap Trigger(t_1) \neq \emptyset$ and $Target(s_n) = Target(t_m)$.

Execution Model

In order to prove properties of the triggering graph it is necessary to define the rule execution model[56]. Rule evaluation is triggered by the following changes in the instance graph:

1. *Attribute-set*: an attribute of an existing object has been modified,

2. *Relationship-set*: the membership of a relationship between two objects has been modified (members added and/or removed),

3. *Object-create*: a new object instance has been created, resulting in the triggering of attribute-set on all attributes, and relationship-set on all object relationships.

4. *Object-remove*: an existing object has been removed. The effect is to remove the object from all relationships triggering relationship-set.

4.4.2 OSL_0 Triggering Graph

The spreadsheet model described in the previous section can be directly mapped into the operations of OSL_0. Given a JSpoon schema C and an OSL_0 propagation rule $s \in S$ of the form "`context` c `:` t `:=` OSL_0-*expression*" then $Target(s) = c.t$ and $Trigger(s)$ is computed by parsing the right-hand-side expression s, and performing an in-order traversal of the tree. Based on the type of the node:

1. Attribute a or relationship r: add to $Trigger(s)$,

2. Literal: no action performed (immutable value),

3. Operation (arithmetic, navigation, or collection): no action performed (continue recursing).

Construction 1 (OSL_0 Triggering Graph) The triggering graph expresses the static dependencies between OSL_0 rules in the context of an object-relationship schema. If S is a set of rules in L labeled $s_1 ... s_n$, the *triggering graph* of S is a directed graph (digraph) $G(V, E)$ constructed as follows:

1. For each rule $s \in S$:

(a) Create a graph node for $Target(s) = c.a$ labeled "$c.a$" (if one does not exist)

(b) for each variable $t \in Trigger(s)$:

i. If t is an attribute $c.a$ then create an graph node labeled $c.a$ (if one does not exist), and add a directed edge $c.a \rightarrow t$ from the trigger to the target (if one does not exist) labeled "s",

ii. Otherwise t is a relationship between c_1 and c_2 identified by the endpoints $c_1.r_1$ and $c_2.r_2$ then create a graph node labeled "$c_1.r_1 : c_2.r_2$" (if $c_1 < c_2$)) or "$c_2.r_2 : c_1.r_1$" (if $c_1 > c_2$) where classes are compared under dictionary ordering. Create a directed edge $r \rightarrow t$ from the relation node to the target node (if one does not exist) labeled "s".

Construction 2 (OSL_0 Triggering Graph Propagator) A triggering graph *propagator* is a label on a directed triggering graph edge identifying the relationship dependency path from the source to the destination node. Propagators are computed by performing an inorder traversal of each rule's right-hand-side expression and marking the relationship path traversed. Propagators are constructed as follows:

1. For each rule $s \in S$ perform an in-order traversal of the right-hand-side rule expression:

(a) For each expression tree node, based on the type of the node:

- dot "`.`" and arrow "`->`" operations:
  - evaluate left node,
  - evaluate right node on left node context.
- name: resolve the name in the context, and add to the propagator.
- binary operation:

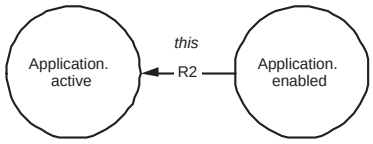

Figure 4.5: OSL_0 assignment graph:
context `Application : active := enabled`

- mark current propagator,
- evaluate left node,
- reset propagator,
- evaluate right node,
- reset propagator.

Examples

Figure 4.5 shows the result of applying the OSL_0 triggering graph construction on Rule 2. The left-hand-side of R_2 assigns the result of the right-hand-side expression to attribute `active` of class `Application`. The right hand-side expression consists of an attribute `enabled` of the same class which generates a directed edge, labeled `R2`, from `Application.enabled` → `Application.active`. The propagator is the default context "`this`", since no relationships are traversed.

It should be noted that the full JSpoon class name, consisting of the package name followed by a dot '.' and the class name, is used to label the nodes. For simplicity, and without loss of generality, the examples shown do not use the JSpoon packaging capability.

Relationship navigation establishes dependencies between the target class and the right-hand-side of the propagation rule. Figure 4.6 shows the graph generated by Rule 4, on the left, and the parse tree of the right-hand-side expression, on the right. The right-hand-side expression parses to the dot "." operator as

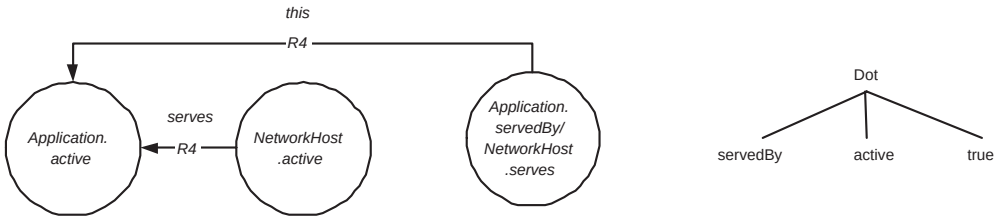

Figure 4.6: OSL_0 relationship graph and parse tree:
context `Application :`
`active := servedBy.active` **default** `false`

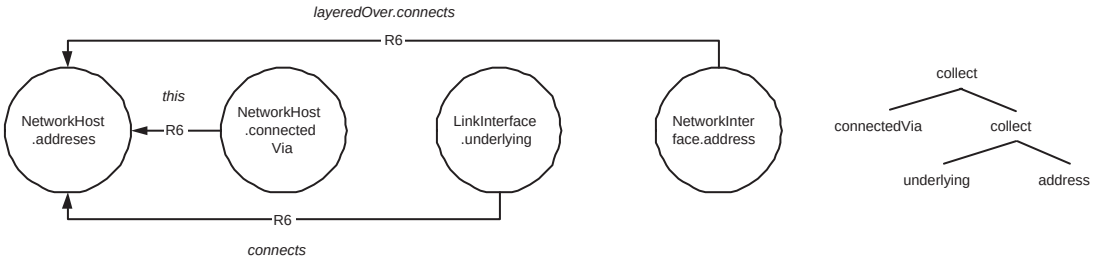

Figure 4.7: OSL_0 collect graph and parse tree : **context** `NetworkHost:`

```
addresses :=
  connectedVia->collect(underlying)
    ->collect(address)->toArray()
```

applied to the `servedBy` to-one relation to obtain an instance of `NetwokHost` and access the `active` attribute. The default value when the relationship is undefined is the literal (constant) `true` which cannot change and is therefore not represented in the triggering graph. The propagator on the `NetworkHost.active` property is the relationship `serves`.

The graph of a more complex OSL_0 statement using the `collect()` operator is shown in figure 4.7. Of interest is the propagation of the `NetworkInterface address` attribute to the `Application active` property by navigating two consecutive relationships layeredOver and connects.

Trigger Graph & Propagator Algorithm

The complete traversal algorithm is listed in table 4.11.

4.4.3 OSL_0 Termination

LEMMA 1 A set of rules in L contains a cycle, if and only if, its triggering graph contains a cycle.

Consider a set S of rules in L containing a cycle. By definition 4, there exists a sequence of rules $s_1, s_2, ..., s_k$ such that $Target(s_1) \in Trigger(s_2)$, ..., $Target(s_{n-1}) \in Trigger(s_n)$, $Target(s_n) \in Trigger(s_1)$. By construction of the triggering graph, the set of arcs $Target(s_1) \rightarrow Trigger(s_2)$, ... $Target(s_k) \rightarrow Trigger(s_1)$ must exist due to the triggering dependencies. Let v_i be any variable $v_i \in Target(s_i)$, and $t_i = Trigger(s_i)$ where $i \leq 1 \leq k$. Then the cycle v_1, t_1, v_2, ... v_k, t_1 will exist.

Conversely, consider the triggering graph constructed out of a set of rules S in the spreadsheet language L. Assume that the triggering graph contains a cycle. Then there exists a vertex v_1 for which with a directed path v_1, v_2,..., v_k, v_1, where $k \geq 2$ because the graph does not contain any self-loops (by construction). Let s_1, s_2, ... , s_k be the labels on the edges $v_1 \rightarrow v_2$, $v_2 \rightarrow v3$, ... $v_k \rightarrow v_1$. Based on the construction, each edge indicates a triggering triggering which is caused by a corresponding rule as shown in table 4.12. The sequence of rules represents a cycle and therefore a cycle in the triggering graph creates a cycle s_1, s_2, ... s_k.

Example An example of a set of expressions whose triggering graph contains a cycle is shown in figure 4.8. The cycle is created by using Rule 2 and adding the following rule:

RULE 23 : **context** InternetRadio: active := enabled

```
Context oslTrigger(Context ctx, OslTreeNode node,
                   TriggerGraph graph) {
  switch (node.type) {
    case PLUS:
      oslTrigger(ctx, node.left, graph);
      oslTrigger(ctx, node.right, graph);
      return(new Context(Number.class, ctx.propagator));
    // case -, *, /, =, <>
    case DOT:
      Context lc = oslTrigger(ctx, node.left, graph);
      return(oslTrigger(lc, node.right, graph));
    case ARROW:
      Context lc = oslTrigger(ctx, node.left, graph);
      switch (node.right.operation) {
        case SIZE:
          return(new Context(Integer.class,ctx.propagator));
        // ... isEmpty(), toArray()
        case COLLECT:
          Context mc = new Context(lc.type.memberType, lc.ctx);
          Context rc = oslTrigger(mc, node.right, graph);
          if (rc.type == Set.class)
            return(rc);
          else {
            return(new Context(Set<rc.type>.class, rc.propagator));
          }
      }
    case NAME:
      Field field = context.type.getField(node.value);
      GraphNode source = graph.addNode(field);
      graph.addEdge(source, ctx.propagator);
      return(new Context(field.type,
                         new Propagator(propagator, field);
    case LITERAL:
      return(new Context(node.type, ctx.propagator));
   }
}
```

Table 4.11: OSL_0 Triggering Graph Construction

Arc	Construction	Rule
$v_1 \rightarrow v_2$	$v_2 \in Trigger(s_1)$	$s_1 : v_2 = f(v_1, ...)$
$v_2 \rightarrow v_3$	$v_3 \in Trigger(s_2)$	$s_2 : v_3 = f(v_2, ...)$
...	...	...
$v_k \rightarrow v_1$	$v_k \in Trigger(s_1)$	$s_k : v_1 = f(v_k, ...)$

Table 4.12: Triggering graph cycle implies rule cycle

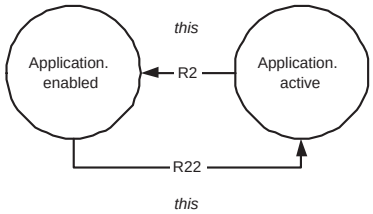

Figure 4.8: OSL_0 same-class cycle example:
context `Application : active := enabled`
context `Application : enabled := active`

Note that this particular cyclical triggering will not result in infinite execution with incremental triggering semantics. However, the behavior of assigning a value to one of the two attributes will be undefined since it depends on the order of rule evaluation.

LEMMA 2 The propagation of a rule set S over an instantiation of an object-relationship schema C can cycle, if and only if, there is a cycle in the rule definitions.

Assume that a propagation cycle over instances of schema C exists. That cycle will be of the form:

$$o_1 \xrightarrow{r_1} o_2 \xrightarrow{r_2} ... \xrightarrow{r_{n-1}} o_n \xrightarrow{r_n} o_1$$

where $o_1...o_n$ are object instances, and $r_1...r_n$ are relations. Let $c(o)$ be the class of the object. Because every relation (edge) in the instance graph must also exist in

the class graph, the cycle:

$$c(o_1) \xrightarrow{r_1} c(o_2) \xrightarrow{r_2} ... \xrightarrow{r_{n-1}} c(o_n) \xrightarrow{r_n} c(o_1)$$

must exist and the schema graph.

Conversely, if there exists a class graph cycle over the rule set S of the form:

$$c_1 \xrightarrow{r_1} c_2 \xrightarrow{r_2} ... \xrightarrow{r_{n-1}} c_n \xrightarrow{r_n} c_1$$

where $c_1..c_n$ are classes, and $r_1..r_n$ relations, then an an instance graph with a cyclical evaluation can be constructed a follows: for each class c_i create an instance o_i. For every relation, r_i instantiate that relation over the two endpoint instances o_j and o_k. Then the path: $o_1 \xrightarrow{r_1} o_2 \xrightarrow{r_2} ... \xrightarrow{r_{n-1}} o_n \xrightarrow{r_n} o_1$ must exist, which forms a cycle.

THEOREM 1 (TERMINATION) The evaluation of a change propagation program consisting of a finite set of OSL_0 change propagation rules will be finite if the program contains no cycles.

Assume that there is an infinite rule evaluation sequence s_1, s_2, s_3, ... Let S be the set of propagation rules in the change propagation program. Because S in finite, the rule evaluation sequence will have to repeat evaluation of some rule s_i. The sequence from the first evaluation of the rule s_i to the next consists of a cycle and therefore invalidates the assumption that the rule set contains no cycles.

Confluence

LEMMA 3 (CONFLUENCE) The order of rule evaluation of an acyclic OSL_0 change propagation program does not effect its final state.

Follows by induction on the number of rules from the fact that every variable must appear as the target of at most one rule. For $n = 1$ the single acyclic rule r cannot trigger itself, and therefore there can only be one sequence (r) length 1. Assume that the rule holds for a program of $n = k$ acyclic rules. Consider a program

of $n + k$ acyclic rules. Assume that there exist two evaluation sequences Seq_1 and Seq_2 which terminate in different states $State_1 \neq State_2$. In that case, there is at least one variable v_i whose value is different. Since the value of variables can only be changed through rule assignment, that variable must appear as the target of a single rule r. By the definition of propagation termination, the rule must have been evaluated in both sequences after the evaluation of its triggers. Therefore the value may be different only because one or more of the triggering variables are have different values. Because the rule set is acyclic rule r will not be re-evaluated an therefore can be removed from the rule set. The new rule-set is of size k which from the assumption cannot generate different states, therefore the assumption is invalid and the induction is proved by contradiction.

4.4.4 OSL_0 Evaluation

DEFINITION 6 (RANK) The *rank* of a propagation rule r, $Rank(r)$, in an acyclic change propagation program is the order of its target variable in the topological sort of its triggering graph.

Any acyclic graph can be topologically sorted in $\Theta(V, E)$ where V is the number of vertices, and E the number of nodes[75]. The algorithm for efficient change rule propagation is listed in table 4.13. The input to the algorithm is an object database *odb* representing the state of the object-relationship model, a triggering graph g for the acyclic change propagation program that is topologically sorted, and the variable t whose value has changed.

Given the changed variable t and the set of changed objects, originally $\{o\}$, the propagation algorithm considers all the rules which may be triggered $s \in S : t \in Triggers(s)$ (line 5). The least ranked rule r is selected (line 8) and its triggering graph propagator is used to select the instances of the rule's target which are effected by the set of changed objects (line 9). The rule is evaluated on each instance (line

```
1  void propagate(RuleSet rules, Object source,
2                 Field field, Value value) {
3    Set changed = assign(source, field, value);
4    if (changed.isEmpty()) return;
5    SortedSet<Rule> pending = new SortedSet<Rule>(rank, '<');
6    pending.add(rules.getTriggeredBy(field));
7    while (! pending.isEmpty()) {
8      Rule rule = pending.removeFirst();
9      for(Object target: rule.collect(changed)) {
10       Object targetValue = oslEval(target, rule.expression);
11       Set propagated = assign(targetValue, rule.target, value2);
12       if (! propagated.isEmpty()) {
13         pending.add(rules.getTriggered(rule.target));
14         changed.add(propagated);
15       }
16     }
17   }
18 }
```

Table 4.13: Change Propagation Algorithm

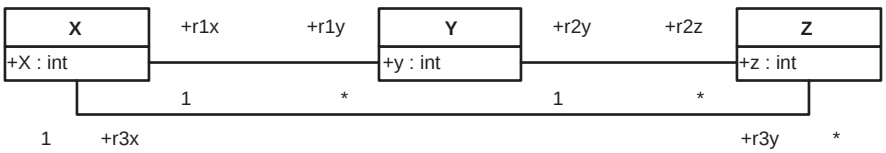

Figure 4.9: Propagation Example Schema

10), and if the target value has been changed, the triggered rules are added to the rule-set in rank-order (line 13), and the changed object added to the changed set (line 14). The loop continues while there are triggered rules to be evaluated.

Example The propagation algorithm is illustrated with an example over the sample object-relationship schema of figure 4.9. The schema consists of three classes X, Y, Z each with one integer attribute correspondingly named x, y, z. Three binary relationships r_1, r_2, r_3 connect class-pairs $\{X, Y\}, \{Y, Z\}, \{X, Z\}$.

Consider the following two rules two rules over this example schema:

```
context Z : z := (r2y.y default 0) + (r3x.x default 0)
context Y : y := (r1x.x default 0) + 1
```

The triggering graph generated for the above two propagation rules is shown in figure 4.10 (R_y denotes the first rule, while R_z the second). As can be readily observed, the graph is acyclic, and therefore can be topologically sorted. The variable graph nodes are shown in topological order from right-to-left. The rank of propagation rules is the order of its target in the topologically sorted triggering graph, so in this case $Rank(R_y) = 2$ and $Rank(R_z) = 3$.

Consider the instantiation of the class graph shown in figure 4.11. The diagram is shown after all rules have been propagated for $x = 1$. For example, the instance z_1 has its property $z = r_2.y + r_3.x = 2 + 1 = 3$. Similarly, instance z_2 has its property $z = r_2.y + 0 = 2 + 0 = 2$.

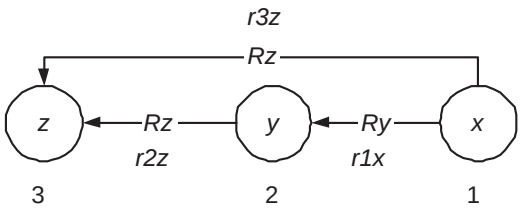

Figure 4.10: Propagation Example Triggering Graph

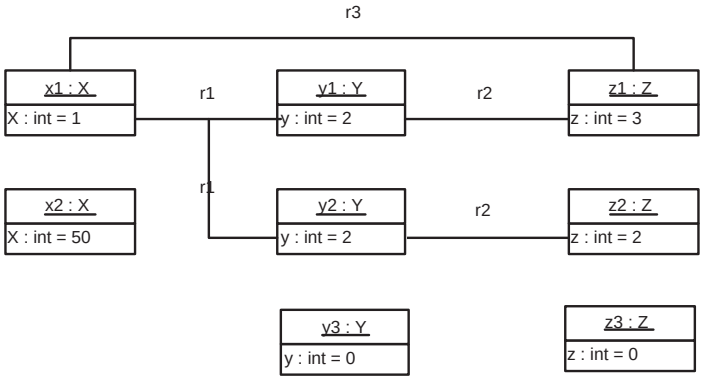

Figure 4.11: Propagation Example Instantiation

Line	Pending Rules	Modified	Rule	$x_1.x$	$y_1.y$	$y_2.y$	$z_1.z$	$z_2.z$
				1	2	2	3	2
1		x_1		**2**	2	2	3	2
6	$R_y(2)$,$R_z(1)$	x_1		2	2	2	3	2
8	$R_z(3)$	x_1	R_y	2	2	2	3	2
9-14	$R_z(3)$	x_1, y_1	$R_y(y_1)$	2	**3**	2	3	2
9-14	$R_z(3)$	x_1, y_1, y_2	$R_y(y_2)$	2	3	**3**	3	2
8		x_1, y_1, y_2	R_z	2	3	3	3	2
8-14		x_1, y_1, y_2, z_1	$R_z(z_1)$	2	3	3	**5**	2
8-14		x_1, y_1, y_2, z_1	$R_z(z_2)$	2	3	3	5	**3**

Table 4.14: Propagation Algorithm Example Trace

Consider a change in the value of attribute x of instance x_1 from $1 \rightarrow 2$ in the object database will require the potential evaluation of change propagation rules. Table 4.14 lists a trace of the propagation algorithm. The propagation algorithm is invoked with the state of the object repository, the topologically sorted triggering graph, the object effected (x_1), the attribute (x) and the value (2) changed.

In line 6 the pending rule evaluation set is initialized to the rules which are triggered from changes in $X.x$, which in this example are R_y and R_z. The set is sorted by the rule rank, and the least-ranked R_y rule is removed in step 8. The rule R_y is evaluated from the instances Y in $x_1.r_{1_y}$, which are y_1 and y_2. Rule R_y is then applied to each instance and both values are updated. The changes in $Y.y$ propagate to $Z.z$, but the rule for R_z is already in the pending queue. Next time through the main loop, rule R_z is selected (line 8), and the instances of Z in $x_1.r_{3_z}$, $y_1.r_{2_z}$, and $y_2.r_{2_z}$ which are z_1 and z_2 are selected for evaluation. Note that instances x_2, y_3, and z_3 where never chosen for evaluation.

4.4.5 OSL_0 Propagation Complexity

LEMMA 4 (TRIGGER RANK) Given any two rules s_1, s_2 in an acyclic propagation program, if rule s_1 triggers another rule s_2 then $Rank(s_1) < Rank(s_2)$

Let s_1 and s_2 be two rules in an acyclic propagation program where s_1 triggers s_2, that is $Target(s_1) \in Trigger(s_2)$. Assume $Rank(s_1) \geq Rank(s_2)$ then by definition of a topological sorted directed acyclic graph, there must be a directed path from $Target(s_2) \rightsquigarrow Target(s_1)$. By construction of the triggering graph, every edge into $Target(s_2)$ is constructed from the $Trigger(s_2)$ set. Therefore, the path must be of the form $v \rightarrow Target(s_2) \rightsquigarrow Target(s_1)$ where $v \in Trigger(s_2)$. Thus $Target(s_2) \in Trigger(s_1)$ which in conjunction with the premise that $Target(s_1) \in Trigger(s_2)$ creates a cycle and contradicts our assumption of acyclicity.

Let V be the number of attributes and relationships in the JSpoon schema where each binary relationship is counted once, and inherited attributes and relationships are counted again for each subclass. Let I represent the number of objects in the repository. Let S be a the number of acyclic OSL_0 rules, and L the max number of operations in a single rule.

LEMMA 5 (RULE EVALUATIONS) Given a single change in the object-relationship model, the propagation algorithm evaluates every rule at most once, and therefore the complexity of the outer loop is $O(S)$.

In every loop execution let s be the rule that is currently being evaluated. Based on the previous lemma, all rules which s triggers must have greater rank, and since the rule set is acyclic, a rule can only be processed once per loop. Therefore the loop is evaluated at most S times.

THEOREM 2 (OSL_0 PROPAGATION COMPLEXITY) The worst-case performance of the change propagation algorithm is $O(S \cdot L \cdot I)$. If the rule size and rule length

$$
\begin{aligned}
&v_1 := f_1(v_2, v_3, ..., v_n) \\
&v_2 := f_2(v_3, ..., v_n) \\
&... \\
&v_{n-1} := f_{n-1}(v_n)
\end{aligned}
$$

Table 4.15: OSL_0 Propagation Algorithm Worst-Case Example

are considered constant values, then the complexity is $O(I)$, that is, propagation grows linearly to the number of instances in the model.

The propagation algorithm loops over the set of triggered rules sorted by rank. Given a change in variable v the set initially contains the rules $s \in S : v \in Trigger(s)$ (line 6). While the set is not empty, the algorithm loops over, retrieving the next rule in the sorted set, evaluating its expression on every effected instance, and potentially assigning a new value to a variable v_2 and adding the statements it triggers $s \in S : v_2 \in Trigger(s)$ into the set.

Within the loop, the complexity of retrieving the first value of a sorted set is $O(1)$ (line 8). The OSL_0 language does not provide any general looping constructs. Evaluating an OSL_0 expression (line 9) involves traversal of an expression tree whose complexity is $O(L)$. The `collect()` operation allows the collation of values from a to-many relation. In the worst case, a relationship may contain all instances of a class, thus each operation may require $O(I)$ processing. Element membership can be established in $O(1)$ through a space trade-off of maintaining a hash-table in addition to the tree structure. Element addition requires $O(\log S)$ but since each rule can only be added once to the set the overall complexity of addition is $O(S \cdot \log S)$.

Therefore, the complexity of the algorithm is $O(S \cdot L \cdot I + S \log S)$. Since the number of objects will be larger than the number of rules the second term can be ignored and the complexity stated as $O(S \cdot L \cdot I)$.

An example of the worst-case rule-set is shown in table 4.15. The length of the dependency and the right-hand-side expressions can be arbitrarily long. In practice such examples never appear in configuration management since the process would be impossible to perform manually.

Rule Updates

Given the set of rules S, the complexity of creating the triggering graph is the cost of parsing the expressions and adding the edges to the graph. The task can be performed incrementally, so that the cost of adding a new rule s is $O(Size(s))$. The topological sort of the triggering graph can be computed in $\Theta(|M|)$[75].

4.4.6 $OSL_{0.5}$ Analysis

The $OSL_{0.5}$ language introduces the `if-the-else` conditional operation. The previous results on OSL_0 termination, confluence, and complexity apply to $OSL_{0.5}$ as well. Termination is not effected because the statement does not introduce looping. Confluence was not dependent on the right-hand-side expression, rather on the fact that only a single assignment statement is permitted per variable. The same holds for complexity, since although the statement may be used to short-circuit computation, in the worst case the conditional expression will not be used, and the complete rule will need to be evaluated.

The main effect of the conditional operation involves the analysis of cycles in the triggering graph. It is possible to define non-trivial cycles in $OSL_{0.5}$ which will never result in an infinite execution. For example the following two rules form a cyclical triggering graph, but no values can satisfy the conditions leading to infinite execution.

```
context Example: a := if (b) c else 1
context Example: c := if (b) 2 else a
```

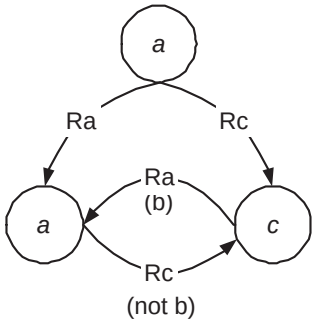

Figure 4.12: $OSL_{0.5}$ Finite Execution Cycle

The triggering graph for the expressions as is shown in figure 4.12. As can be readily observed, the graph contains the cycle $a \rightarrow c \rightarrow a$. In fact, the condition for the existence of the cycle is the satisfiability of the statement (b and not b) which cannot be satisfied for any value of b.

$OSL_{0.5}$ Conditional Triggering Graph

Definition 7 (Phantom Cycle) An $OSL_{0.5}$ change propagation program cycle is called a *phantom cycle* if there is no instantiation of schema S such that change can propagation along the cycle.

Construction 3 (OSL_0 Triggering Graph Conditional Propagator) A triggering graph *conditional propagator* is a propagator whose path is extended with boolean conditions. Change is propagated along each propagator relationship only if the boolean expression evaluates to *true* in the current path context. Conditional propagators are constructed by extending the propagator construction 2 with an additional case for the conditional operation, and redefining the handling of name handling:

- `if-then-else` operation

  - push conditional statement into condition stack
  - evaluate left node
  - pop conditional statement
  - push **negated** conditional statement into condition stack
  - evaluate right node
  - pop conditional statement

- name: resolve the name in the context, and add to the propagator, along with the top of the conditional stack (if any).

Definition 8 (Provably Acyclic Rule Set) An $OSL_{0.5}$ rule set is called provably acyclic if the conjunction of all conditional propagators an every triggering graph cycle is unsatisfiable.

$OSL_{0.5}$ Evaluation

The OSL_0 change propagation evaluation algorithm cannot be directly applied to $OSL_{0.5}$ due to the possible presence of cycles and the fact that a cyclic digraph cannot be topologically sorted. Since a rule may not trigger its own evaluation, in the worst case, an evaluation loop may execute $\sum_{i=1}^{n} n = O(n^2)$ times. If the graph is topologically sorted after each cycle is broken by removing an arbitrary edge, then the average case can be improved.

$OSL_{0.5}$ Complexity

Given an $OSL_{0.5}$ triggering graph containing cycles, the analysis of the conditional propagator conjunction is at least as difficult as boolean satisfiability (SAT) which is a well know problem in NP[76]. If all conditionals are conjunctive boolean expressions of variables then the problem can be solved in polynomial time. The problem

of determining whether even just one single multivariate polynomial equation has an integer solution is undecidable[77]. If the conjunction contains linear equality equations over reals then in can be solved in $O(n^3)$.

In practice, change propagation conditions are simple expressions, otherwise they could not have been manually enforced by human systems administrators. When computing the cycle satisfiability an upper time limit may be placed on computation when new rules are introduced. If the time out is exceeded, the new rule will be rejected as unverifiable.

The complexity analysis for $\text{OSL}_{0.5}$ is similar to that of OSL_0 with the difference that in the worst case, the outer loop may evaluate $O(S^2)$ times and therefore the complexity of the $\text{OSL}_{0.5}$ propagation algorithm is $O(S^2 \cdot L \cdot I)$. The algorithm continues to scale linearly with the number of objects, but the rule evaluation constant will be larger.

4.4.7 OSL_1 Analysis

OSL_1 introduced iteration over sets. This is a limited form of iteration which is guaranteed to terminate if the set is finite. A limited type of iteration was already available in OSL_0 in the form of the `collect` operator. Therefore, the `iterate` operator will not effect the worst-case complexity analysis.

However the combination of the new `allInstances` operator with the `iterate` operator allows users to perform first-order relational calculus operations. Selection, projection, union and intersection were already shown as part of the language. A join may be performed as as shown in the example below which sets the `serves` relationship of a DNS server object to the DNS resolver objects configured to use the server's IP address.

```
context DomainNameServer: serves :=
  DomainNameResolver->allInstances
```

```
->select(r | r.servers->exists(s | s.address = this.address))
```

OSL_1 relational operation techniques can leverage the extensive research into relational database operation optimizations[78].

4.5 Controlling Autonomic Behavior Across Domains

In the last four decades, the scale of computer networks has exploded from the order of tens of nodes in the original ARPANET, to the order of billion of Internet nodes today. Scalability was a major challenge along the path of this growth, forcing the adoption of new protocols and management techniques. Flat-structure protocols such as the HOSTS name-to-address file were replaced by hierarchical protocols such as DNS[3]. Routing was also made hierarchical through adoption of inter-domain routing protocols such as BGP[79]. Human scalability was achieved by assigning the resulting hierarchical domains to network and systems management groups operating at different organization levels.

Initially, most network services operated within individual domains. Examples included file sharing, databases, local name resolution, and client-server applications. This pattern has changed with the increasing popularity of world-wide services such as SMTP, FTP, and HTTP. In addition, the proliferation of Internet Service Providers (ISPs) has meant that many of the clients for local domain services are connected through Virtual Private Network (VPN) connections. Finally, the emerging multi-tiered web-services architectures, such as Sun J2EE and Microsoft .Net, have created new and complex dependencies between network services.

Todays reality is that that domains can no longer be isolated at the network (IP) layer, with the addition of a few isolated global distributed services such as

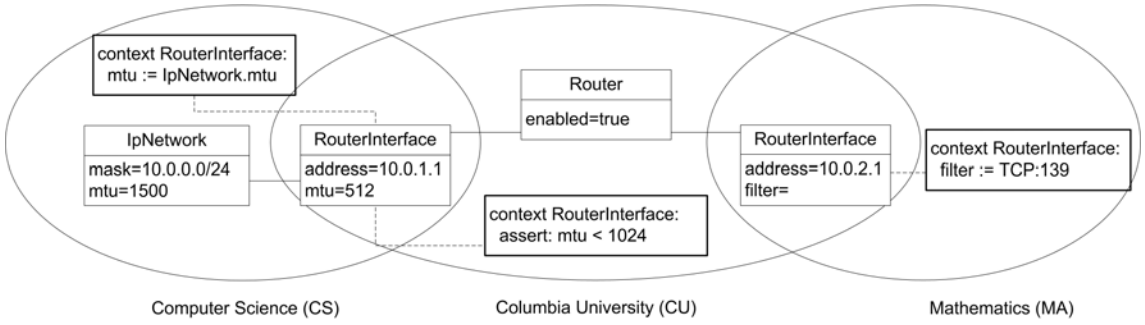

Figure 4.13: Management Domains

DNS. Complex service dependencies cross existing domain structures creating reliability, optimization, and security challenges that stress the current manual management work-flows. Any approach to network element automation must therefore provide solutions that address these issues.

4.5.1 Inter-Domain Rule Propagation Automation

The previous section covered the analysis of propagation rules within a single administrative domain. In order for change to propagate between two domains, there must be one or more management objects that are shared by both domains. For example, a departmental domain will share the IP router interface with the organizational domain. Changes to the status of the router interface will effect the departmental domain, and may also propagate to the organizational domain.

An example of such shared objects is shown in figure 4.13. The figure depicts three organizational domains representing Columbia University (CU) and two of its academic departments, Computer Science (CS) and Mathematics (MA). Each department is responsible for its own administration. However, the management of its peering points with the University is shared with the campus network management service group.

Today, due to limitations in access and concurrency control, such "shared" resources are only jointly managed in principle. In practice, management is maintained by the service provider, and changes propagating from the client are manually propagated through some type of request ticketing system.

In order to apply the spreadsheet change propagation approach to automating inter-domain propagation, it must be possible to analyze and restrict the scope of automated propagation in a scalable manner. The simple approach of analyzing the union of all propagation domain rules is not practical because it will not scale in terms of rule evaluation. Moreover, different domains are likely to enfonce different policies ot their internal resources, and the union operation will result in multiple rules for the same class target. Finally, it is unlikely that domains will want to export their change rules to others, since it may expose security vulnerabilities.

Construction 4 (Summary Trigger Graph) Let S be a set of OSL propagation rules over an object-relationship schema C, and $C_e \subset C$ represent classes whose instances may be exported to other domains. If G is the triggering graph for S over C, then the summary triggering graph G_e for S over C_e is constructed as follows:

- for each node $n \in G$ representing a field in a class $c \in C_e$ add n to G_e
- for each node $n \in G_e$ and for every node $o \in G_e$ (including n) such that there is a path $n \overset{G}{\rightsquigarrow} o$ add a directed edge $n \overset{G_e}{\rightarrow} o$ in G_e.

An example summary graph construction is shown figure 4.14. The example shows two domains from the previous example, CS and CU, and two attributes of the shared class `RouterInterface`. Within the CS domain, a propagation path exists from `RouterInterface.type` → `IpNetwork.mtu` → `RouterInterface.mtu`. Because the path spans classes that are not shared, it will be included in the summary graph as an abbreviated edge from `RouterInterface.type` → `RouterInterface.mtu`

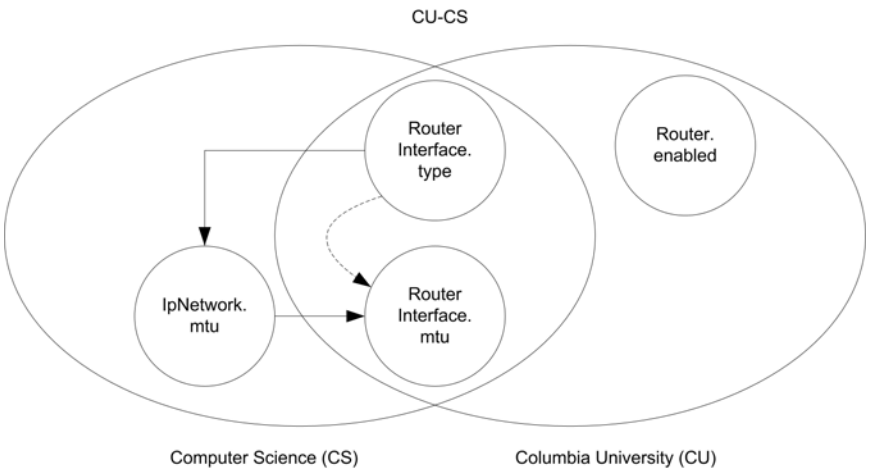

Figure 4.14: Management Domain Cycle Analysis

(shown as a dotted arrow).

LEMMA 6 A triggering graph will contain a cycle containing one or more exported class attributes if and only if there is a cycle in the summary triggering graph.

Assume there is a cycle in the triggering graph containing one or more exported class attribute. The cycle can be expressed as:

$$v_1 \xrightarrow{s_1} v_2 \xrightarrow{s_2} \ldots \xrightarrow{s_{n-1}} v_n \xrightarrow{s_n} v_1$$

where $s_1 ... s_n$ are class attributes or relationships and $s_1 ... s_n$ are propagation rules. By the assumption there exists an attribute or relationship s_i whose class $c \in C_e$ is exported. By construction, if there are no other exported attributes or relationships in the path, then there should be a summary path from $s_i \rightarrow s_i$. It can be easily shown by induction that this will hold for any number of exported path nodes.

If the summary triggering graph G_e contains a cycle, then either all path edges are in the regular triggering graph G, or the cycle contains edges which represent a path in G. Substituting the summarized edges with their G path produces a cyclical path in G.

Inter-Domain Cycle Analysis

CONSTRUCTION 5 (SHARED DOMAIN TRIGERRING GRAPH) Given a set of propagation domains $\{D_1, D_2, ...D_n\}$, let $\{C_{e_1}, C_{e_2}, ...C_{e_n}\}$ be the corresponding export schemata. Construct the summary triggering graph for pair (D_i, C_{e_i}). The shared domain triggering graph is constructed by the union of all triggering graphs.

The shared domain triggering graph is used to perform cycle analysis and to determine rule ranking that is used for rule evaluation ordering. The shared domain defines the types of permitted propagation, and can be the subject of policy rules. For example, a laptop connected to a foreign network may contain policy rules to abort propagation of DNS resolver configuration via DHCP. Such propagation has been used in the past to hijack network outgoing connections for monitoring. Similarly, a domain may allow route propagations, but filter those which advertise routes to addresses that are within the domain.

4.6 Previous Work

4.6.1 Spreadsheets

The spreadsheet change propagation model was first applied to grid-based financial applications[80]. Financial spreadsheets operate over cells and do not provide object abstractions or structured relationships. Popular spreadsheet applications, such as Microsoft Excel, perform run-time cycle analysis and therefore they are not well suited to handling unsupervised data feeds. For example, table 4.16 demonstrates a set of accepted propagation rules which are potentially cyclical base on the spreadsheet data instantiation. Excel will only signal an error if `A4 = FALSE`.

A1	= if ($A3, $A2, 1)
A2	= if ($A4, 2, $A1)
A3	TRUE
A4	TRUE

Table 4.16: Microsoft Excel Runtime Cycle Checking

```
EMPLOYEE.DEPTNAME := 'Research'
$FIND ANY EMPLOYEE USING DEPTNAME
while DB_STATUS = 0 do
  begin
  $GET EMPLOYEE;
  writln(EMPLOYEE.FNAME, " ", EMPLOYEE.LNAME);
  $FIND DUPLICATE EMPLOYEE USING DEPTNAME
  end;
```

Table 4.17: CODASYL Relational Data Model Program

4.6.2 Network Data Model

The network data model[81] preceded the relational data model and was employed in several commercial database products. The network data model consists of sets of typed records containing data items. Set elements are connected via one-to-many relationships. Access to the data model is performed through a data access language that is embedded in a general purpose programming language such as COBOL or PASCAL. Table 4.17 shows an example taken from [19]. Statements preceded by the '$' symbol represent database operations to differentiate from PASCAL constructs.

The network data model shares many common features with the object-relationship model. The access language, however, was embedded into an general purpose imperative language, making static analysis impossible. This chapter introduces a restricted access language over an object-relationship model with spreadsheet model semantics that can be statically analyzed at rule declaration time.

4.6.3 Event Correlation Systems

The object-relationship model has been previously used to model network event propagation in SMARTS[57]. The object-relationship event model provides means to analyze and correlate events associated with objects and the the events are assumed to be generated through sources external to the system. Change propagation rules, in contrast, provide means to effect changes in the very objects and are assumed to be generated as integral components of the system. InCharge does not incorporate means to statically validate non-cyclical propagation, of central importance in change propagation.

4.6.4 Event-Condition-Action/Active Database Systems

Active database systems[82] automate the propagation of changes through the definition of Event-Condition-Action (ECA) rules. A typical ECA rule is structured as a statement of the form ¡event, condition, action, precedence¿ where event is some database update (table row insertion, removal, modification), the condition is an SQL Boolean expression, the action consists of an SQL statement, and the optional precedence provides an ordering of rule evaluations. Changes in active databases propagate over relations established via relational table joins.

Prior active database research has explored algorithms for analyzing termination, and confluence (rule evaluation order dependency)[56]. Termination in active rule sets has been verified through cycle-detection in a triggering graph. The triggering graph is formed by creating a vertice for each rule and a directed edge from each rule to the rules that it can potentially trigger. Analysis of the trigger dependencies between rules depends on the expressiveness of the condition and action language.

The work presented in this thesis differs from previous work on ECA/active database systems in two important aspects:

1. The object-relationship (OR) model provides a different data modeling language and methodology that is more appropriate for expressing the type of propagation paths engineered in networked systems. The OR-model with its support for inheritance, and stored relations as object attributes provides a more natural method to express dependencies that the database relational model. Although it is possible to map the OR-model into a relational database schema, the result would include a significantly expanded number of tables, and the ECA statements would become very complex.

2. The graph constructed by the invention differs from the triggering graph approach in that it represents object class attributes as nodes as opposed to the rules. The expression language used in the invention is restricted so as to support static determination of cycles and ambiguities, as opposed to potential cycles and ambiguities.

3. The spreadsheet acyclic propagation semantics allow are easier to reason with for rule authors, since they express the target values as a result of a function, rather than as a response to a change.

4.6.5 Feature Interaction

Feature interaction is a well recognized problem in system integration, and has been studied extensively in the telecommunication industry[83, 84]. Telecommunication systems typically operate as large finite state machines and the emphasis of that work has been to analyze the behavior of the state machines, as opposed to the effects of their configuration. The work in this chapter differs in that it focuses on the configuration plane, rather than the data/algorithm plane, and represents interactions as propagations over an object-relationship model.

4.6.6 Work-flow Systems

Research on Work-flow Management Systems (WMS)[85] has focused on connecting systems in work-flows as "black-boxes" with little or no control over internal configuration and state. The approach of this thesis has been to develop an integral instrumentation model of network elements, and to use that model to compute the acyclic propagation of changes over relationships.

4.6.7 Constraint Propagation Systems

The Constraint Satisfaction Problem (CSP) has been studied extensively in a variety of applications[45, 46]. Previous work on constraint-based management has been pursued[47, 48]. The focus of these projects has been on employing constraints for the diagnosis of network faults and on algorithms for constraint satisfaction. The challenge in such systems is to express constraints in a manner that does not over-constraint (no solution), or under-constraint (too many solutions leading to non-determinism). The spreadsheet model is a hybrid Event-Condition-Action/Constraint system in which propagation rule expressions can be thought of as constraints on object attribute value, while assignment creates side-effect based propagation. It should be noted that the autonomic architecture supports use of a CSP solver as a knowledge layer plugin.

4.6.8 Change Propagation Systems

Previous work on change propagation over relationship has been pursued in [86, 87, 88]. The DSM language[86] is used to express fixed, link-by-link, propagations over relations, and is very restricted in its expressive power. The AMP propagation language[88] is used to define relational Event-Condition-Rules that propagate over an object relationship model. The language does not admit simple analysis and one

is not presented.

4.7 Conclusion

Autonomic behavior refers to the ability of systems to perform react to changes in order to maintain operation. Current systems are designed to minimize change propagation since such operations have to be performed by human administrators. Automation using existing programming techniques has met with limited success due to challenges in configuration access, concurrency, recoverability, and automation feature interaction.

This chapter has introduced the spreadsheet acyclic change propagation model and the OSL language for expressing spreadsheet rules over an object-relationship configuration model. The three increasingly powerful languages presented support the expression of common configuration change propagation rules, without resorting to use of a Turing-complete language. The OSL languages have been statically analyzed for cyclical definitions, and for determining optimal rule evaluation sequences. The rule evaluation algorithm uses the information compiled from the static class model in order to maintain propagation in the much larger instance model. The final result presented introduces a hierarchical domain approach to controlling the propagation of changes across domains in a scalable manner.

Chapter 5

Autonomic Applications

5.1 Introduction

The peer-to-peer autonomic management technologies presented in the previous chapters of this thesis has been implemented in a large research prototype called NESTOR. The prototype includes a distributed transactional object modeler, an object-relationship modeling definition language compiler, an incremental OSL change rule and OPL policy constraint interpreter, as well as adapters for different management protocols and elements. The NESTOR modeling language supports the JSpoon management attribute declarations, but is not embedded in Java. NESTOR was developed in two successive versions which provided practical experience with different automation architectural designs.

The NESTOR prototype has been demonstrated in several management automation applications. NESTOR was first demonstrated in automating the propagation of of DHCP address assignments into the DNS hosts. The application supported user mobility without requiring new specialized protocols. The NESTOR prototype was also applied to the management of security in dynamic networks. The application demonstrated automated adaption of link layer and application

layer security configuration to the presence of new network users. In the context of Active Networks[89] NESTOR has been demonstrated in automating configuration of network virtualization[90], and active multimedia adaption technologies. Then NESTOR prototype has also been deployed on the Active Networks Backbone (ABONE) network for instrumentation of Active Nodes, EEs and AAs, NESTOR has been released to researchers at at Telcordia Technologies as a platform for developing a distributed firewall based on security policies[91]. Finally, NESTOR has been demonstrated in a web-service mobility demonstration, involving DNS and web-server reconfiguration in response to server mobility.

This chapter continues with a section outlining NESTOR prototype features and technologies. The next two sections present two of the NESTOR automation demonstrations in greater detail.

5.2 NESTOR Prototype Overview

An initial prototype of the NESTOR system was built using the MODEL language and InCharge repository provided by SMARTS[12]. The prototype employed the Event-Condition-Action (ECA) rules (which can be compiled from declarative constraints). The prototype was used to demonstrate Internet node plug & play functionality .

Experience with this first prototype helped guide the current NESTOR design. The complexity of coding both the access mechanisms and the schema mapping operations led to the addition of the protocol adapter layer. ECA rules quickly proved to be difficult to manage even with a small number of high-level constraints defined. It was discovered that more than one rule was required to support a single constraint, and that it was not uncommon to write simple definitions that would lead to cycles in execution. Declarative expressive constraints are used in the current design and are safely compiled internally to ECA style rules.

The second NESTOR prototype has been written in Java using Sun's Jini infrastructure[20]. In this prototype the RDS exports its services using a Java Remote Method Invocation (RMI) interface. The remote RDS interface enables managers to create distributed transactions, and perform object operations (lookup, create, destroy). Application layer management applications employ the Jini lookup and discovery mechanism for obtaining a reference to the remote RDS service object. When an application invokes the create transaction method on the remote RDS interface, the RDS server returns an object implementing the Jini transaction interface. Internally, the Java RDS prototype implements the Jini transaction manager interface and semantics for performing distributed two-phase commits.

Management application object lookups occur in the context of a transaction and return a proxy object implementing the same interfaces as the ones of the requested repository objects (maintaining in addition a lock on the requested objects). Unlike the real repository objects, proxy objects contain copies of the configuration values and do not propagate changes to the managed element, even though all method invocations are stored in the transaction log. Proxy object references are initially returned as "hollow" objects whose values are retrieved from the repository at the time of the first object access.

Constraints and rules are first class objects. When the management application commits an update transaction, the modeler invokes the constraint and propagation manager with a reference to the transaction log. The manager analyzes the log and determines the OSL rules that need to be reevaluated, and proceeds to compute and assign their values. As part of that process additional rules may need to be re-evaluated. This process is guaranteed to terminate since cyclical rule definitions are disallowed. Once all rules have been evaluated, the constraints are asserted. If all constraints are maintained the manager commits the transaction (the constraint manager is a member of every repository update transaction) the

logged updates will be applied to the real repository objects in the order in which they were made. In cases where the same attribute has been updated multiple times due to the execution of change rules, only the last update is written.

Other components of the prototype system include the model compiler and constraint and rule interpreter. The model compiler transforms MODEL interface definitions into Java interface definitions. As part of the transformation, MODEL attribute declarations are mapped to pairs of set/get methods, and relationships are converted into references to classes implementing the OSL collection semantics. In the current version, the constraint and rule expressions are stored in string form instead of being translated into a Java language method. The constraint and propagation manager contains a built-in OSL interpreter that evaluates each expression when detecting a possible violation. Future versions will explore the performance gains of compiling OSL expressions, and optimizing the triggering of constraint and rule evaluation.

Adapters supported in the current NESTOR prototype include Linux (interfaces, routing, processes, firewall rules), SNMP (MIB-II), CISCO IOS (switch VLAN and interfaces, router interfaces, routes and firewall rules), Virtual Active Networks[90] (VAN), and Anetd[92]. In addition, the prototype includes a graphical browser supporting navigation and manipulation of the NESTOR repository, as well as visualization of layer-2 topology.

The current implementation also supports security features including user authentication, fine-grained capability and access control-based authorization, as well as connection encryption. Authentication and connection encryption are based on the SSL/TLS protocol. The SSL X.509 certificates are associated with a first-class user object instances that are belong to one or more groups. Groups are assigned repository-wide permissions such as connect, search, subscribe, lock, etc. Associating permissions with each object, or adding capabilities to the user or group

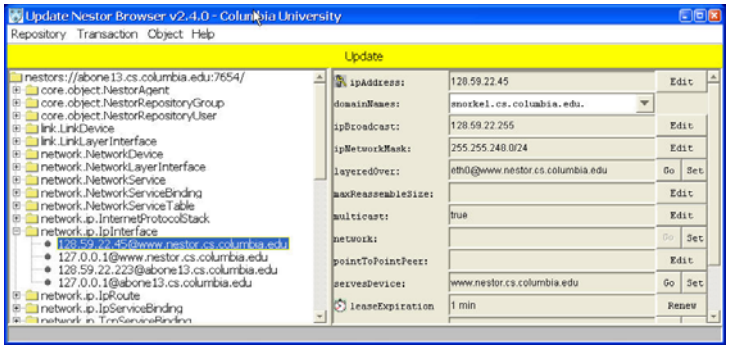

Figure 5.1: NESTOR Modeler Graphical Browser

objects controls object-level permissions. Examples of object-level permissions are get, set, shared-lock, exclusive-lock, delete and may be associated with all or some specific object attributes.

Because user, group and permission objects are first class objects, NESTOR constraints and propagation rules may operate on them. As a result, it is possible to affect dynamic configuration of security permissions. For example, it is possible to award a user with special permissions on all hosts that are physically co-located with the machine in which he/she is logged on the console. Note that such general rules can only be expressed thanks to the unified configuration model.

A screen-shot of the NESTOR prototype browser is shown in figure 5.1. On the left panel, the browser displays a tree whose first level are repositories, the second level includes the list of available RDL interfaces, and the third layer contains object instances. The browser subscribes for notification of class loading/ unloading events on each repository. When a class node is first expanded, the browser subscribes for object create/remove events on the particular class. If the user selects an object instance, it is displayed on the right panel. In this particular example, an Anetd instance is shown. The object is read in a caching transaction, that does not obtain any locks. The browser (NESTOR client) is notified of any

changes to cached objects. Relations can be navigated by clicking on the "Go".

5.3 Managing Security in Dynamic Networks

Consider a scenario in which Jane Consultant, who is employed by Corporation A, is visiting a client in Corporation B. During her meeting, Jane realizes that she needs to access files in her home directory which have not been copied onto her laptop. When plugged into her home network in A, Jane simply clicks an icon on her desktop to access her files. What are her choices while plugged into B's network? She can establish a slow but potentially insecure modem connection to Corporation A (over a wireless connection for example, or perhaps the phone call is routed over the Internet). Alternatively, she can plug her laptop into an Ethernet port within Corporation B; assuming she gets connected at all, it will likely be a window-less connection to B because Corporation A may not open X services in its network to hosts outside. Neither method offers access comparable to what Jane would get within her home network.

What makes the problem more challenging is that the two networks are separately administered, with independent security policies. For example, Corporation A might filter certain services when the user is plugged into a remote network. Corporation B might require that guest machines not be able to send or receive traffic directly from any machine within B's network, and that guest machines may only access remote VPN nodes. If there is a way to provide access without violating either company's policy, we would like all necessary reconfigurations to be automatic and not require manual intervention. Of course, if there is no way to provide access without violating one security policy or another, Jane cannot be provided this service. The difference in our approach is that we have a language designed to express these concerns explicitly at a high level as policies and mechanisms to support the semantics of these policies by appropriately reconfiguring the network.

A number of configuration changes are necessary to provide Jane access when the security policies allow it. In our example, some of the changes involve the Dynamic Host Configuration Protocol (DHCP)[2] server, switches and firewalls in B, and firewalls, file servers and encryption protocols within A, and decryption in Jane's laptop.

5.3.1 The Experimental Testbed

Recall that in our scenario Jane, whose home is corporate network A, is now connected to company B's network and wants Web/E-mail/Telnet access to files in her company A. To simplify the exposition, we make a few simple assumptions about the two networks. These assumptions are not necessary in practice; the approach is more general that the example we choose to illustrate the capabilities of our management platform. In particular, let us suppose that Company B uses a switched network which supports Virtual LANs (VLAN) and that company A's firewall supports Virtual Private Networks (VPNs) in order to provide remote access to its users over the Internet. Our actual implementation uses Linux for firewalls and hosts and Cisco switches for VLAN support. Further details of the equipment used are provided later in this section.

Requirements for the Scenario

In order to achieve transparent access to the services that Jane wants from her laptop the following configuration changes are necessary:

1. An available Ethernet port will need to be located and Jane's laptop physically connected (in the premises of company B).

2. The laptop will need to be configured for the local network environment, including parameters such as IP address, netmask, DNS servers, default gate-

ways, SOCKS servers, etc. Ideally, this will be achieved automatically using DHCP.

3. In order to maintain Company B's security policy, the laptop will either have to be connected to a special "guest" network and the switch port must be configured for a "guest" virtual LAN, or all the internal services must be guaranteed to require authentication. The last option is exceedingly difficult to implement in typical networks today and are, in fact, the main motivation for using firewalls to protect corporate networks.

4. Depending on the configuration setting of Company B's firewall, the laptop's address may have to be explicitly allowed to initiate outgoing connections. Company B's security policy may require disabling the laptop's access to any external sites other than Company A since it holds an IP address in Company B's domain. This can be achieved by limiting external connections to the VPN protocol.

5. Once the laptop can reach the Internet, it will need to establish a Virtual Private Network (VPN) connection with Company A's firewall whose policy may be to grant remote hosts limited access to internal resources. Such policies will need to be enforced by all internal services in Company A's network.

Network Topology

The two networks are shown in Figures 5.2 and 5.3. For simplicity, Company A's network consists of the internal subnet 172.16.1.0/24 (net-1) and the VPN subnet 172.16.6.0/24 (net-6). VPN clients are allocated addresses from the second network (net-6). The internal Linux NFS server is configured dynamically by the company-A NESTOR server to restrict mobile user access to their home directory. A Linux

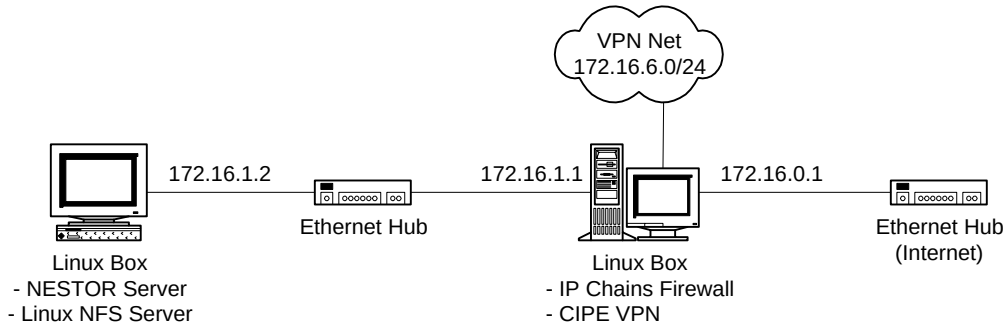

Figure 5.2: Company A Network

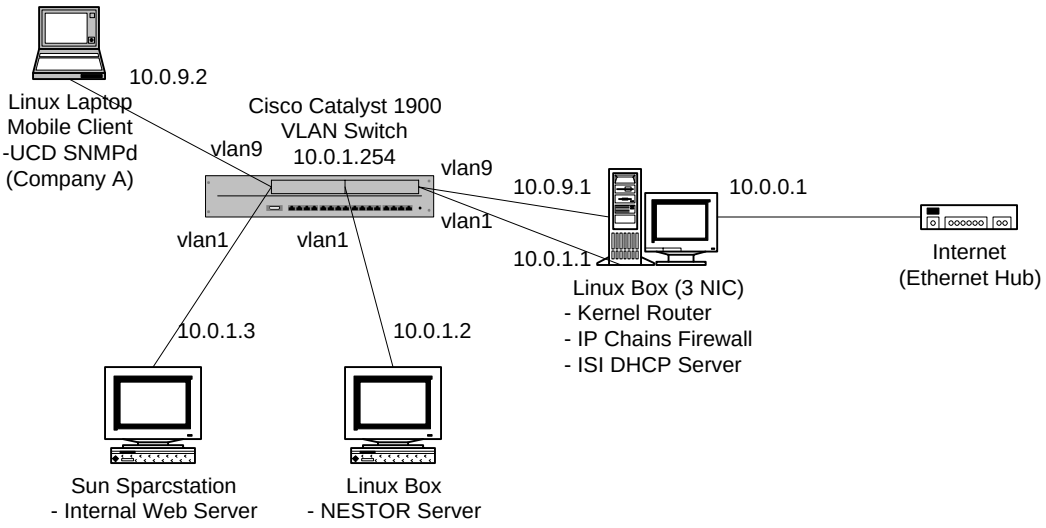

Figure 5.3: Company B Network

workstation is used to provide routing, firewalling, and VPN services for the network of company A. The route table is statically set with paths to the internal network, the VPN network, and company B's network. Firewall rules are added to deny all incoming traffic access to the internal network of company A (using Linux Kernel IP Chains). Finally, the CIPE[93] Linux software is configured to enable the remote establishment of a VPN tunnel.

The network of company B is slightly more involved. Instead of an Ethernet hub, the internal network is a switched network. Layer-2 switching is provided by a Cisco Catalyst 1900 with support for Virtual LANs (VLAN). The VLAN switch provides for the physical separation between trusted and untrusted IP nodes mandated by the policy of company B. Vlan port assignments will be handled dynamically by the NESTOR server. Company B also has a trusted subnet 10.0.1.0/24 (net-1) and untrusted subnet 10.0.9.0/24 (net-9). Virtual LANs with IDs 1 and 9 physically separate traffic to net-1 and net-9 respectively. A Linux workstation provides routing, firewall and DHCP services to the internal network. The router has three interfaces, one connected to vlan-1/net-1, another connected to vlan-9/net-9 and the last connected to the external network. Static routes are configured to route traffic from the internal networks (net-1 and net-9) to the external network. Firewall rules prevent any incoming traffic from the external network to the internal network with the exception of established TCP connections to net-9 (guest network). Furthermore, a firewall rule restricts outgoing connections from net-9 exclusively to the VPN port of external network destinations. Obviously, no routing is configured between net-9 and net-1. The DHCP server on the host is configured to listen to the two net-1 and net-9 interfaces for DHCP requests. All unknown hosts are allocated IP addresses from net-9, while a list of trusted hosts (based on their unique client ID which may be their Ethernet address) are set to be allocated addresses from net-1. It is assumed that the list of trusted client identifiers

is supplied to the NESTOR server. These trusted identifiers will be dynamically configured into the DHCP server by NESTOR.

The networks of the two companies were connected through a common Ethernet hub, with static routes configured between the gateways. In this experiment, the NESTOR servers in each company operate independently of each other. The details on how dynamic configuration of the aforementioned networks occurs are discussed in the next section.

Constraints and Automatic Reconfiguration

We now outline how Jane gets transparent access without violating the security policy of either network. The policy of company B not to allow guest machines to gain access to the internal (trusted) network can be translated into the following topology-dependent constraints on device and service configuration. (The next section describes how these constraints can be expressed formally in the NESTOR configuration language).

- Switch ports to which trusted hosts hosts are connected must belong to the internal (net-1) VLAN. Switch ports to which guest (or unknown) hosts are connected must be configured for the guest VLAN (net-9). In cases where internal and guest hosts are connected to the same switch port, the port should be set to the guest VLAN (an alternative would be to disable the port). Violation of this predicate is handled by reconfiguring the VLAN membership of the offending port.

- A firewall rule should prohibit any traffic (including the guest network) from entering the internal network (except for established TCP connections).

- The DHCP server should allocate internal IP addresses only to trusted hosts.

Additionally, company B policy further restricts guest Internet access by limiting guest connections to remote VPN servers. The goal of this policy is to prevent guests from attacking or misusing other networks while in ownership of a company B IP address. This policy is translated into a simple configuration constraint limiting all outgoing traffic from the guest network to well-known ports of VPN services.

The security policy of company A states that remote (VPN) users should receive limited services. In this example, remote users are restricted to accessing only their home directory. We map this policy (there are other ways) to file server configuration by initially denying all directory mounts by VPN hosts. When users sign in with the VPN server to obtain an IP address, permission is added for the particular host to temporarily mount the user's directory. The constraint on file server configuration states that VPN hosts should only be allowed to mount the directory whose user is logged on to the VPN server with that address. This constraint spans the configuration of the VPN server as well as the file server.

In the overall scenario, these constraints (predicates and actions) achieve dynamic reconfiguration in the following manner :

1. User Jane cannot find an available Ethernet port for her laptop, so she "borrows" the network connection of an existing host (thereby obtaining physical access to the internal network).

2. Jane's laptop requests configuration information from the DHCP server,

3. The DHCP server returns a lease on a guest network IP address since the MAC address of Jane's laptop is not included in the DHCP daemon internal network list (at this point the laptop's IP configuration is inconsistent with the link-layer network to which it is connected).

4. By polling the Ethernet switch, the company B NESTOR server discovers

the new laptop host. The constraint manager evaluates the constraints which might be violated by the connection of a new host to the switch. The security constraint will be found to be violated, since an unknown (therefore untrusted) host is connected to the internal network. The action component of the constraint is executed, resulting in the reconfiguration of the affected switch port VLAN. Note, that if Jane had connected to an unused switch port, which by default is assigned to the guest network, this constraint would not have been violated. Future prototypes will include mechanisms other than polling.

5. Jane's laptop is now connected to the guest network and will attempt to establish a VPN connection with its home server (at company A) (any other access request, such as web access, is filtered by the firewall of company B).

6. Once the laptop has authenticated with company A's VPN server it is assigned a virtual IP address.

7. The NESTOR server of company A detects this new lease (by polling for the configuration state of the VPN server). The constraint on the file server configuration will be violated since the file server will not be configured to allow home directory mounts from that IP address. This violation will be handled by adding the IP address of the VPN host to the access list of the user's home directory on the file server.

8. After completing her work, Jane disconnects from the VPN server. Again, the company A NESTOR server detects this change, and re-evaluates the affected constraints, resulting in the removal of the file access permission for the previously allocated VPN IP address.

9. After Jane disconnects her laptop from the company B network, and graciously reconnects the host whose connection she had "borrowed", the com-

```
interface companyB::EthernetVlanSwitch {
  relationshipset consistsOfPorts, EthernetVlanSwitchPort, partOf;
  attribute IpInterface ipInterface;
}

interface companyB::EthernetVlanSwitchPort : CompanyB::Node {
  attribute boolean isEnabled;
  readonly attribute int portNumber;
  attribute int vlanId;
  relationshipset forwardsNodes, EthernetNode, forwardedBy;
  relationship partOf, EthernetVlanSwitch, consistsOfPorts;
}

interface companyB::SecurityManager {
  attribute boolean isTrusted;
  relationshipset manages, Node, securityManager;
}
```

Table 5.1: Network Model Example

pany B NESTOR server will detect this event and the ensuing constraint violation, leading to the reassignment of the affected switch port back to the internal VLAN.

5.3.2 Network Model and Configuration Constraints

The first step in using NESTOR for our experiment is modeling the network. This section gives some examples of model definitions for network B. Models are expressed in the MODEL language[57] which is an extension of the CORBA[94] IDL with support for relationships, and other features useful for event correlation. Table 5.1 shows a subset of the model definitions for company B.

Consider the `EthernetVlanSwitchPort` interface definition. This interface models the configuration of a port in an Ethernet switch supporting VLANs. The MODEL definition states that the interface is part of the `companyB` package and inherits from the `Node` interface. Three attributes are declared to model: the state

of the switch port (enabled/disabled), the port number (a read-only value), and the integer ID of the Virtual Lan to which the port is assigned. The relationship definitions declare a many-to-one relation mapping the port to its enclosing switch, and a one-to-many relation associating the port with the Ethernet (layer 2) nodes which are actively connected to the port.

The `EthernetVlanSwitchPort` interface represents a device-independent configuration model for an Ethernet switch port supporting Virtual LANs (VLAN). In order to instantiate such an object in the NESTOR repository, an implementation of that interface must be provided with support for the configuration protocols of the actual device being modeled. The next step will therefore be to compile the MODEL interface definitions into a target implementation language. The current NESTOR prototype is built in the Java language and the model compiler converts the extended IDL interface definitions to a set of Java interfaces. As part of the compilation, attribute definitions are converted to a pair of get/set methods (one for read-only attributes) following a simple design pattern. Relationships are compiled into references to collections implementing the OSL (Object Spreadsheet Language) collection semantics.

The Ethernet switch supporting VLANs used in this experiment was a CISCO Catalyst 1900 with enterprise edition firmware. The Catalyst supports several SNMP MIBs and may also be configured using a menu system as well as from the command-line. The Bridge SNMP MIB `dot1dTpPortTable` table was used to instrument the `consistsOfPorts` attribute of the Catalyst `EthernetVlanSwitch` implementation. The implementation class registers with the NESTOR SNMP adaptor to receive notification of updates to the table. When a new port is detected, a new instance of the `CiscoCatalyst1900Port` class is constructed. The port `forwardsNodes` relationship is instrumented through the Bridge MIB `dot1dTpFdbTable`. See [95] for details of the components of these tables. The VLAN ID attribute is

```
context EthernetVlanSwitchPort: inv:
  isEnabled and isTrusted(vlanId)
    implies forwardNodes->forAll(n | n.isTrusted)
```

Table 5.2: A Declarative Constraint: Trusted ports should only forward frames of trusted nodes

instrumented using the Cisco IOS adaptor parameterized by the command sequence appropriate for obtaining the VLAN id of this port. The `CiscoCatalyst1900Port` class, implementing the `companyB.EthernetVlanSwitchPort` interface, was defined with a single constructor parameterized by the IP address of the managed switch, the switch port number, and an SNMP and IOS authentication object. The authentication objects encapsulate protocol-specific security access information such as passwords and certificates.

Programming Constraints

Based on this model of the network, constraints are defined to maintain the security policies of each domain. To take an example, consider the constraint caused by company B's policy that untrusted hosts should not have access to the internal network. This policy is translated into several constraints on the configuration of network devices. For example, a constraint on the switch states that trusted ports (i.e., those configured for a trusted VLAN) must only be connected to trusted hosts. This constraint, expressed in the OCL language, is shown in table 5.2. In this example all instances of the `EthernetVlanSwitchPort` which are enabled and whose VLAN ID is assigned to a trusted LAN, are required to only be connected to trusted hosts.

Self-management is achieved by programming a change rule which maintains this constraint. For example, violation of the above constraint, that trusted ports should only forward frames for trusted nodes, may be handled by programming

```
context EthernetVlanSwitchPort:
  vlanId := if (isTrusted) and
              (forwardNodes->exists(n | not n.isTrusted)) then
            trustedID
          else
            untrustedID;
```

Table 5.3: Switch VLAN ID propagation rule

the computation of switch port VLAN IDs based on port membership, as shown in figure 5.3

Populating the Repository

The NESTOR repository may be populated manually or using a graphical user interface that can generate objects given the model and the appropriate parameter values. The repository is accessed through a Java Remote Method Invocation API. The API supports methods for adding and removing objects, locating objects based on the class and attributes, and initiating transactions. To add an object to the repository a systems administrator initiates a transaction and then adds the object within the transaction. The object must implement one or more model interfaces and support the serializable interface (i.e. may be stored as a byte string for transport over the network). Storage in the repository is provided on a lease basis which must be renewed by some entity such as a lease renewal manager, or the object itself. If a lease expires an object may be killed or archived if possible. If there are constraints that may be affected, an error message is raised on the console. This obviates the need for vigorous garbage collection. The current NESTOR prototype utilizes the Jini[20] distributed leasing, event, and transaction APIs.

The repository can also be populated with the help of a utility for topology discovery. The utility executes on a host, and periodically pings each network

address to establish a map of active nodes[1]. Currently, our topology manager accepts classless IP network and netmask combinations. Once a node is detected as being active, the utility attempts to extract information using the SNMP protocol, and tests for service availability by attempting to connecting to different services (such as Telnet, HTTP, NFS, FTP, etc). In its current incarnation, the topology manager is mostly focused on discovering workstations (such as Linux and Windows NT boxes) and supplying information about their interface configurations, route tables, and active services.

Returning to our scenario, the administrator constructs a new instance of the `CiscoCatalyst1900Switch` object using the IP address assigned to the management interface of the VLAN switch and the appropriate authentication information for administering the switch which are the SNMP community and IOS passwords. NESTOR repository objects implement an initialization and control interface (analogous to Java applets) so that their execution can be controlled by the NESTOR server. An object may query the repository for services such as adapters using an instance of the `RepositoryContext` interface. For example, the switch object will use an SNMP and IOS adaptor instances. New adaptors may also be used provided they implement the `NestorAdaptor` interface). After obtaining the necessary adaptor references, the object will subscribe for notification of changes in the relevant SNMP objects, and IOS results.

When the administrator commits the transaction to create the new switch object, the transaction manager will verify that the addition did not violate any constraints. The constraint, shown in table 5.2, may be violated when new instances of objects implementing the `EthernetVlanSwitchPort` interfaces are created. Assume the initial switch state does not violate the aforementioned constraint. When user Jane connects her laptop computer to network B, and in particular to

[1]The security warnings that these may generate will have to be handled.

a switch port, the switch SNMP bridge MIB table `dot1dTpFdbTable` will add the laptop's MAC address. At the time of the next poll by the NESTOR SNMP adapter, the change will be detected resulting in notification of the subscribing `CiscoCatalyst1900Switch` object. The switch object will look up for an instance of `EthernetNode` with the same MAC address, creating a new instance if one is not found. The `EthernetNode` is then added to the switch's `forwardsNodes` relation. At the point where all propagated changes have been reflected in the model, the switch object will commit the changes. At this point the constraint manager will again verify the set of constraints which may have been affected by the transaction. In this example, since Jane connected her laptop to a switch port previously assigned to the internal network, the constraint on switch port VLAN state will be violated, and the policy script will be executed as outlined earlier in the paper.

5.4 Active Networks

The Active Networks Daemon (Anetd)[92] is currently being used to deploy and manage EEs on the DARPA ABONE[96]. This section will present the automation of Anetd configuration using NESTOR.

5.4.1 Anetd Data Modeling

The first step in managing a resource in NESTOR is to identify the relevant model classes. In the case of Anetd, a class will be associated with each Anetd process, and will be related to processes (EEs) owned by ABONE users, and executing on the Internet ABONE host. The standard NESTOR model contains class abstractions for Internet hosts, users and processes. The Anetd-specific classes will be expressed in the Resource Definition Language (RDL) as extensions to the base model and the AN-related classes shown in table 5.4. In this example, fragments of two interfaces

```
interface anetd::Anetd: system::Application {
 attribute String version;
 attribute boolean isPrimary;
 relationshipset manages, AnetdProcess, managedBy;

 // Also: port, javaVM, childPort, ...
}

interface anetd::AnetdProcess: anets::ExecutionEnvironment {
  relationship managedBy, Anetd, manages;
  attribute boolean isPermanent;

  // Inherits: anepID, servedBy(Node), serves(AA)
  //
  // Also: filePreloadURL, workDirectory
  //       isAutoKill, standardInputFile, ...
}
```

Table 5.4: Anetd daemon and process RDL definitions

```
context anets::ActiveNode
  servesApplication
  ->select(app: System::Application |
           app.oclIsKindOf(anetd::Anetd))
  ->select(ad : anetd::Anetd | ad.isPrimary)
  ->size = 1
```

Table 5.5: Exactly one primary Anetd per Active Node (OPL constraint)

are shown in table 5.4, one for modeling an Anetd process (Anetd), and another for the EEs hosted (AnetdProcess).

Note that the service provided by Anetd is partially, but not fully, that of the Node OS. Therefore, instead of extending the anets::ActiveNode class we establish a new "manage" relation between an AnetdProcess and an Anetd instance. An AnetdProcess object inherits the generic Execution Environment functions, and extends them with Anetd-specific parameters, such as the work directory, and whether the process is permanent (persistent across restarts).

5.4.2 Anetd Semantic Modeling

Once the RDL data model has been designed, the model author may add intrinsic constraints and propagation rules expressed in the Object Policy Language (OPL) and the Object Spreadsheet Language (OSL). For example, it may be stated that there should be exactly one primary Anetd within each 1 ActiveNode. Similarly, a propagation rule may state that if the Java installation changes in the ActiveNode, this should be propagated to the configuration of the local Anetd objects. A OPL example for the former constraint is shown in table 5.5. It states that for each instance of ActiveNode, identify the Anetd processes it is hosting, and assert that exactly one of these processes is primary.

In case of failure of the primary Anetd, the above constraint will be violated (size = 0). NESTOR enables automated recovery from such failures via a propagation rule that restarts the failed Anetd daemon, or assigns a new primary. The propagation rule shown in table 5.6 performs the latter by selecting the process with the lowest port number to act as primary. The rule operates by identifying the Anetd objects in each ActiveNode, sorting them by port number, and assigning the lowest numbered one be the primary.

It is also possible, that the failure of a non-primary Anetd process will break the forwarding chain. Recovery from such inconsistent states can also be automated via a propagation rule. The propagation rule shown in table 5.7 sorts the list of Anetd objects in and sets the `childPort` attribute to the port of the previous Anetd daemon. Note that both rules shown must agree on the election process, that is, that the sort is based on the port number. However, the order in which the rules are applied is not important since there are no cyclical dependencies. The NESTOR propagation manager checks for cyclical definitions and rejects such rules, similarly to spreadsheets.

```
context anets::Anetd
  isPrimary := partOf.serves
               ->select(a | a instanceof Anetd)
               ->sortBy(a | port, Integer.LESS_THAN)
               ->first() == this
```

Table 5.6: Primary Election (OSL propagation rules)

```
context anets::Anetd
  childPort :=
    let daemons = partOf.serves
                  ->select(a | a instanceof Anetd)
                  ->sortBy(a | port, Integer.LESS_THAN)
    in
      if (daemons->first() == this)
        0
      else
        daemons->get(indexOf(this) - 1).port;
```

Table 5.7: Forwarding chain (OSL propagation rules)

5.4.3 Anetd Adapter

Once the data and semantic models have been defined, an adapter must be provided that will instrument Anetd processes as objects in the repository. In particular, this adapter must support bi-directional instrumentation, with read as well as write capabilities.

The adapter functionality may be integrated into the Anetd source code, by embedding NESTOR directory management API functions, or may be provided externally via some polling or publish-subscribe mechanism. By embedding the NESTOR model into the service it is possible to obtain fast response to changes, with the lowest polling overhead. It is also possible to avoid storing any persistent configuration data by taking advantage of the NESTOR repository persistence capabilities. In many cases, such as proprietary hardware and software, it may not be possible to modify the service itself. In such cases, the adapter must be executed as

an external process that polls and sets the configuration of the service using some external protocol. Examples include adapters using protocols such as SNMP and LDAP, those simulating terminal input such as CISCO IOS adapters, and those parsing and modifying configuration files, such as an HTTPd adapter.

The Anetd adapter developed in this example will be external and will utilize the Anetd SC[92] control protocol. The protocol supports remote polling and configuration of Anetd processes. Ideally, the adapter will provide its own native implementation of the SC protocol client. An alternative would be to use the existing Anetd distribution SC binary client as a system process and then read its console text output. In either case, the external daemon must be able to send SC queries and parse their response so that they can be mapped to the appropriate NESTOR model class instances.

At startup time, the adapter will have to discover the NESTOR repository (RDS) where the objects will be instantiated. In the current NESTOR system, the location of the repository may be either configured, or discovered using Jini[20] discovery. The choice of which repository to use is an open issue that is currently being investigated.

Once the repository has been discovered, the adapter will first have to look for existing objects that may fully, or partially represent the managed resources. For example, the agent will have to lookup the object for the ActiveNode that should have been instrumented by the NodeOS adapter. Also, the agent will have to check for any objects that it has created previously, whose lease has not expired. In order to perform these lookups, the adapter must identify key attributes that can uniquely identify the relevant model objects. For objects previously created, it is possible to use the agent's unique ID. Currently, there are open issues relating to model composition and they are being investigated. The discovery, lookup, poll and apply process is illustrated in figure 5.4.

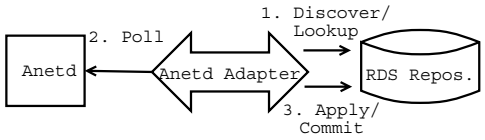

Figure 5.4: Anetd Read Instrumentation

The adapter then enters a polling loop in which any changes in the underlying resources are applied and committed to the model. It should be noted that because the Anetd SC remote interface is not transactional, it is not possible to determine if the polled state represents multiple real-world threads of change, or that these threads have completed execution. Therefore, the adapter is forced to lump all detected changes into a single transaction. If the transaction is aborted due to a constraint violation, all changes will need to be rolled-back.

Due to the lack of locking mechanisms in most configuration protocols, it is also possible that the configuration of a real element may change in the process of performing or committing a transaction. This may occur in cases where systems administrators bypass the NESTOR system in changing configuration, or due to some dynamic element reconfiguration. Mechanisms for addressing this issue by stating requirements on concurrent managed resource access are being investigated. It should be noted, that changes occurring through the repository API are always transactional and therefore can always be isolated and controlled.

The second function of a NESTOR adapter is to propagate changes initiated by NESTOR application-layer transactions, or change propagation rules. The process is illustrated in figure 5.5. Once a transaction is committed to the model, that is, all propagation rules have been applied, and all constraints have been verified, the repository checks if any of the effected objects are instrumented by an agent. A special to-one relation with an agent object indicates that changes to the object need to be propagated. In the Anetd example, changes to the instrumented objects,

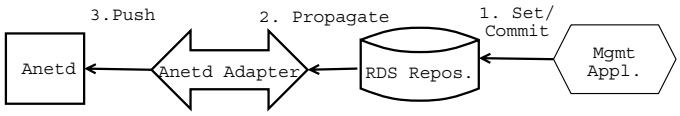

Figure 5.5: Anetd Set Instrumentation

such as instances of Anetd and AnetdProcess will be collected and transmitted to the adapter. The adapter will use this information to effect the necessary changes by issuing SC requests. In addition to attribute value modifications, the log will contain relation modifications. For example, to deploy a new EE, a user may create an AnetdProcess object, set its values, and then add that object to the "manages" relation of an Anetd object. When this change log is sent to the adapter, it will interpret this action as a request to deploy a new EE. Similarly, an EE may be terminated through its removal from the "manages" relation.

An alternate scenario would involve the failure of the primary Anetd process on the ActiveNode. The adapter would detect this failure and the corresponding object would be removed in a repository transaction. Before the transaction was committed, the propagation rule from table 5.6 would be fired, and an alternate Anetd process would be selected as a primary. Assuming that no constraints were violated, the transaction would be committed and the changes propagated to the adapter, and then to the actual Anetd processes. It should be noted that reliance on the SC protocol means that an Anetd instance cannot be configured if it is unreachable or in a stopped state. In such cases, the ActiveNode adapter may be used to kill the Anetd process and start a new one.

Authoring of agents is a labor-intensive and potentially complex process. It is envisioned that in the future services will support standardized configuration languages (such as XML[97]) and some form of transaction and event based configuration protocol that will provide better support for identifying real-world threads

of change. The NESTOR system provides additional library support for performing common agent tasks, such as a minimal merge of two object graphs (polled to repository objects), and automated agent creation for popular protocols such as SNMP. For example, it is possible to provide a generic SNMP adapter that can be configured with a mapping between MIB tables and objects, and OIDs and object attributes.

Chapter 6

Conclusion

The high cost of operating current networks is a result of limitations in the design of existing network management architectures. Attempts to provide automation as a layer over existing architectures will fail because they cannot satisfy the concurrency control requirements of automation processes. Current network management operations will therefore not be capable of sustaining the growing number of networked devices, and complex dependencies created by new web-based service architectures.

The proposed peer-to-peer organization offers significant advantages over the traditional manager-agent (client-server) organization. The unified model allows managers to discover, access and manipulate the configuration of all network elements. Transactional access to management information creates an environment supporting safe multi-manager access, as well as recoverable configuration change semantics. The unification of the traditional roles of manager and element allows management functions to be distributed in different elements, supporting autonomic behavior. Transactions establish natural policy enforcement points, and can be used to more accurately correlate the root cause of network failures or inefficiencies. Distribution of element configuration creates a scalable management infrastructure, which can continue to operate under network partition, to maintain

policy, and effect self-healing.

The JSpoon approach to autonomic management offers several substantive advantages over current alternatives. Management information is consolidated with element design and can be maintained through its life-cycle evolution. Instrumentation, data models, knowledge model and their bindings can be generated and managed through compiler support and static-time validation. Network events can be intercepted synchronously to constrain and extend the behavior of objects. Knowledge modules can be seamlessly incorporated with elements by vendors of autonomic computing products, independently of the element vendors enabling synergistic evolution of products. instrumentation, data and knowledge models can be unified across multiple elements greatly simplifying the task of providing autonomic self-managing capabilities of large composite systems.

Automation using existing programming techniques has met with limited success due to challenges in configuration access, concurrency, recoverability, and automation feature interaction. The spreadsheet acyclic change propagation model provides a simple computation model. The OSL language for expresses spreadsheet rules over an object-relationship configuration model, without being Turing-complete. OSL rule sets can be statically analyzed for cyclical definitions, and for determining optimal rule evaluation sequences. The rule evaluation algorithm uses the information compiled from the static class model in order to maintain propagation in the much larger instance model. Policy and change-management can be scaled using a hierarchical domain approach to controlling the propagation of changes.

6.1 Future Work

It is envisioned that the P2P repositories will be highly distributed to support scalable operation as well as recovery during failures. Future research will determine

the granularity of distribution (service, node, LAN, department), and the location of repositories for non-programmable devices, such as hubs, switches, and COTS routers. Distribution of the repositories and adapters will require merging of partial models. For example, a switch adapter may discover an Ethernet node identified by a unique MAC address and proceed to generate a simple EthernetInterface object. Later, an adapter may be provided for the host, and the interface may be recognized as an EncryptingEthernetInterface supporting hardware based datagram encryption.

In some cases it may also be possible to infer relations, such as co-location, based on information collected from multiple elements. A transitive closure operation will be investigated as an OSL_1 extension.

Financial spreadsheets support a visual rule development environment in the form of cell highlighting, and cycle visualization. The results of the static OSL analysis may be used to design equivalent visual management configuration interfaces. Such tools will need to support hierarchical domain visualization, and path summarization.

Network provisioning involves a planning in which the network model is applied to a set of physical resources. Further research is needed to develop self-provisioning mechanisms and to investigate their dynamic behaviors for network scenarios.

Bibliography

[1] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *IEEE Computer*, vol. 36, no. 1, 2003.

[2] R. Droms, "Dynamic Host Configuration Protocol," Tech. Rep. RFC 1531, IETF, 1993.

[3] P. Mockapetris, "Domain names - implementation and specifications," Tech. Rep. RFC 1035, IETF, November 1987 1987.

[4] B. Ronen, M. A. Palley, and H. C. Lucas, Jr., "Spreadsheet analysis and design," *Commun. ACM*, vol. 32, no. 1, pp. 84–93, 1989.

[5] M. Nicolett, K. Brittain, and P. Adams, "Enterprise management ROI and cost reduction in 2003," tech. rep., Gartner, nov 2002.

[6] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, and D. Orchard, "Web services architecture," Tech. Rep. WD-ws-arch-20030808, W3C, 2003.

[7] ISO, "Information processing systems - open systems interconnection - basic reference model - part 4: Management framework," Tech. Rep. 7498-4, ISO, 1989.

[8] J. Case, M. Fedor, M. Schoffstall, and J. Davin, "A Simple Network Management Protocol (SNMP)," Tech. Rep. RFC 1067, IETF, 1988.

[9] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 4, no. 3, pp. 382–401, 1982.

[10] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM (JACM)*, vol. 32, no. 2, pp. 374–382, 1985.

[11] A. Dupuy, S. Sengupta, O. Wolfson, and Y. Yemini, "Netmate : A network management environment," *IEEE Network Magazine (special issue on network operations and management)*, 1991.

[12] SMARTS, *InCharge*. White Plains, NY, 1997.

[13] ISO, "OSI Common Management Information Protocol (CMIP)," Tech. Rep. ISO/IEC 9596-1, CCITT Recommendation X.711, ISO, 1988.

[14] A. Pell, K. Eshgi, J. J. Moreau, and S. Towers, "Managing in a distributed world," in *Fourth IFIP/IEEE International Symposium on Integrated Network Management*, 1995.

[15] S. Judd and J. Strassner, "Directory-Enabled Networks : Information model and base schema," Tech. Rep. Version 2.0.2-2, DEN Ad Hoc Working Group, 1998.

[16] OMG, "Meta object facility (mof) specification," Tech. Rep. Version 1.3, Object Management Group (OMG), 1999.

[17] OMG, "Unified Modeling Language (UML)," tech. rep., Object Management Group (OMG), 1997.

[18] A. Schade, P. Trommler, and M. Kaiserswerth, "Object instrumentation for distributed applications management," in *IFIP/IEEE International Conference on Distributed Platforms*, pp. 173–185, 1996.

[19] R. Elmasri and S. B. Navathe, *Fundamentals of Database Systems*. Addison-Wesley, 2 ed., 1994.

[20] Sun Microsystems, "Jini architecture specification," tech. rep., Sun Microsystems, 1998.

[21] Sun Microsystems, "Universal plug and play (UPNP," tech. rep., Sun Microsystems, 1998.

[22] G. Banavar, M. Kaplan, R. E. Strom, and D. C. Sturman, "Information flow based event distribution middleware," in *ICDCS Workshop on Electronic Commerce and Web-Based Applications*, 1999.

[23] Sun Microsystems, "Java Message Service (JMS)," tech. rep., Sun Microsystems, 2002.

[24] H. Hazewinkel, C. Kalbfleisch, and J. Schoenwaelder, "Definitions of managed objects for WWW services," Tech. Rep. RFC 2594, IETF, May 1999.

[25] S. B. Davidson, H. Garcia-Molina, and D. Skeen, "Consistency in a partitioned network: a survey," *ACM Computing Surveys (CSUR)*, vol. 17, no. 3, pp. 341–370, 1985.

[26] J.-L. Lin and M. H. Dunham, "A survey of distributed database checkpointing," *Distributed and Parallel Databases*, vol. 5, no. 3, pp. 289–319, 1997.

[27] P. A. Bernstein and N. Goodman, "The failure and recovery problem for replicated databases," in *Proceedings of the second annual ACM symposium on Principles of distributed computing*, 1983.

[28] J. K. Kim and G. G. Belford, "A protocol for failure and recovery detection to support partitioned operation in distributed database systems," in *Proceedings of 1986 fall joint computer conference on Fall joint computer conference*, pp. 1189–1196, 1986.

[29] J. Moy, "OSPF version 2," Tech. Rep. RFC 2328, IETF, 1998.

[30] UCD, *NET-SNMP*, 2003. http://net-snmp.sourceforge.net/.

[31] H. Takagi, *Analysis of Polling Systems*. MIT Press, 1986.

[32] G. Goldszmidt and Y. Yemini, "Distributed management by delegation," in *The 15th International Conference on Distributed Computing Systems*, (Vancouver, British Columbia), IEEE Computer Society, 1995.

[33] IETF Distributed Management (disman) Charter, "Distributed management framework and services," in *IETF 1996 Proceedings*, Dec. 1996.

[34] T. G. Griffin and G. Wilfong, "An analysis of BGP convergence properties," in *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, 1999.

[35] E. C. Jr., Z. Ge, V. Misra, and D. Towsley, "Network resilience: Exploring cascading failures within BGP," in *Proceedings of the 40th annual Allerton Conference on Communications, Computing and Control*, 2002.

[36] S. Sengupta, A. Dupuy, J. Schwartz, and Y. Yemini, "An object-oriented model for network management," in *Object Oriented Databases with Applications to CASE*, Englewood Cliffs, NJ: Prentice-Hall, 1991.

[37] F. Teraoka, Y. Yakote, and M. Tokoro, "A network architecture providing host migration transparency," *Computer Communication Review*, vol. 23, no. 4, 1991.

[38] Y. Yemini, A. Dupuy, S. Kliger, and S. Yemini, "Semantic modeling of managed information," in *Second IEEE Workshop on Network Management and Control*, (Tarrytown, NY), 1993.

[39] Distributed Management Task Force (DMTF), "Common Information Model (CIM) specification," Tech. Rep. Version 2.2, DMTF, June 1999.

[40] S. K. Goli, J. Haritsa, and N. Roussopoulos, "ICON: A system for Implementing Constraints in Object-based Networks," in *Integrated Network Management, IV*, 1995.

[41] J. e. Widom and S. e. Ceri, *Active database systems: triggers and rules for advanced processing*. San Francisco, CA: Morgan Kaufmann, 1996.

[42] M. Harlander, "Central system administration in a heterogeneous unix environment," in *8th USENIX System Administration Conference (Lisa VIII)*, 1994.

[43] J. Finke, "Automation of site configuration management," in *11th USENIX System Administration Conference (Lisa '97)*, 1997.

[44] J. Abbey and M. Mulvaney, "Ganymede: An extensible and customizable directory management framework," in *LISA XII*, (Boston, MA), 1998.

[45] E. C. e. Freuder and A. K. e. Mackworth, *Constraint-based reasoning*. MIT Press, 1994.

[46] E. Tsang, *Foundations of Constraint Satisfaction*. Academic Press - Harcourt Brace & Company, 1993.

[47] M. Sabin, A. Bakman, E. C. Freuder, and R. D. Russel, "Constraint-based approach to fault management for groupware services," in *International Symposium on Integrated Network Management (IM'99)*, (Boston, MA), 1999.

[48] M. Sabin, R. D. Russel, and E. C. Freuder, "Generating diagnostic tools for network fault management," in *The Fifth IFIP/IEEE International Symposium on Integrated Network Management (IM'97)*, (San Diego, CA), 1997.

[49] L. Wall, T. Christiansen, R. Schwartz, and S. Potter, *Programming Perl*. O'Reilly & Associates, 2 ed., 1996.

[50] D. Mills, "Simple Network Time Protocol (SNTP)," Tech. Rep. RFC 2030, IETF, 1996.

[51] Sun Microsystems, "Java Management eXtensions instrumentation and agent specification (v.1.2)," tech. rep., Sun Microsystems, 2002.

[52] B. Lampson and H. Sturgis, "Crash recovery in a distributed data storage system," tech. rep., Xerox, Palo Alto Research Center, 1976.

[53] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[54] T. Berners-Lee, R. Fielding, U. Irvine, and L. Masinter, "Uniform Resource Iidentifiers (URI): Generic syntax," Tech. Rep. RFC 2396, IETF, 1988.

[55] L. Van Der Voort and A. Siebes, "Termination and confluence of rule execution," in *2nd International Conference on Information and Knowledge Management (CIKM 93)*, (Washington, DC), 1993.

[56] A. Aiken, J. M. Hellerstein, and J. Widom, "Static analysis techniques for predicting the behavior of active database rules," *ACM Trans. Database Syst.*, vol. 20, no. 1, 1995.

[57] D. Ohsie, A. Mayer, S. Kliger, and S. Yemini, "Event modeling with the model language : A tutorial introduction," tech. rep., SMARTS (System Management Arts), 14 Mamaroneck Ave., White Plains, New York, 10601, 1996.

[58] S. A. Yemini, S. Kliger, E. Mozes, Y. Yemini, and D. Ohsie, "High speed and robust event correlation," *IEEE Communications*, May 1996.

[59] R. Enns, "XMLCONF Configuration Protocol," Tech. Rep. draft-enns-xmlconf-spec-00, IETF, 2003.

[60] M. Rose and K. McCloghrie, "Structure and identification of management information for tcp/ip-based internets," Tech. Rep. RFC 1065, IETF, 1988.

[61] M. Garschhammer, R. Hauck, H.-G. Hegering, B. Kempter, M. Langer, M. Nerb, I. Radisic, and H. Rlle, "Towards generic service management concepts - a service model based approach," in *7th IFIP/IEEE Symposium on Integrated Management (IM 2001)*, (Seattle, WA, USA), pp. 719–732, 2001.

[62] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbinger, G. Johnson, M. Modvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf, "An architecture-based approach to self-adaptive software," *IEEE Intelligent Systems*, vol. 14, no. 3, pp. 54–62, 1999.

[63] M. Mansouri-Samani and M. Sloman, "Monitoring distributed systems," *IEEE Network*, vol. 7, no. 6, pp. 20–30, 1993.

[64] Y. Yemini, A. Konstantinou, and D. Florissi, "NESTOR: An architecture for NEtwork Self-managemenT and ORganization," *IEEE JSAC*, vol. 18, no. 5, 2000.

[65] A. Konstantinou, Y. Yemini, and D. Florissi, "Towards self-configuring networks," in *DARPA Active Networks Conference and Exposition (DANCE)*, IEEE Press, 2002.

[66] L. Ricciulli, P. Porras, P. Lincoln, P. Kakkar, and S. Dawson, "An adaptable network control and reporting system (ancors)," in *DARPA Active Networks Conference and Exposition (DANCE)*, (California, USA), 2002.

[67] D. Garlan and B. Schmerl, "Model-based adaptation for self-healing systems," in *ACM SIGSOFT Workshop on Self-Healing Systems (WOSS'02)*, (Charleston, S.C.), pp. 27–32, 2002.

[68] I. Georgiadis, J. Magee, and J. Kramer, "Self-organizing software architectures for distributed systems," in *ACM SIGSOFT Workshop on Self-Healing Systems (WOSS'02)*, (Charleston, S.C.), 2002.

[69] N. Almasri and S. Frnot, "Dynamic instrumentation for the management of EJB-based applications," in *Systmes composants adaptables et extensibles*, (Grenoble, France), 2002.

[70] S. DaSilva, *Netscript: A Language System for Active Networks*. PhD thesis, Columbia University, 2002.

[71] OMG, "Object Constraint Language specification (OCL)," Tech. Rep. ad/97-08-08 (version 1.1), Object Management Group (OMG), September 1, 1997 1997.

[72] Y. Leontiev, M. T. Özsu, and D. Szafron, "On type systems for object-oriented database programming languages," *ACM Computing Surveys (CSUR)*, vol. 34, no. 4, pp. 409–449, 2002.

[73] R. W. Sebesta, *Concepts of Programming Languages*. Addison-Wesley, 3rd ed., 1996.

[74] UPNP Forum, "JavaDoc 1.4 Tool Documentation," Tech. Rep. http://www.upnp.org/, UPNP Forum,, 2003.

[75] D. E. Knuth, *The Art of Computer Programming: Fundamental Algorithms*, vol. 1. Addison-Wesley, 1973.

[76] S. A. Cook, "The complexity of theorem-proving procedures," in *Proceedings of the third annual ACM symposium on Theory of computing*, pp. 151–158, 1971.

[77] Y. V. Matiyasevich, *Hilbert's Tenth Problem*. MIT Press, 1993.

[78] G. Graefe, "Query evaluation techniques for large databases," *ACM Computing Surveys (CSUR)*, vol. 25, no. 2, pp. 73–169, 1993.

[79] Y. Rekhter and T. Li, "A border gateway protocol 4 (BGP-4)," Tech. Rep. RFC 1771, IETF, Mar. 1995.

[80] D. Bricklin, "VisiCalc," tech. rep., Lotus Corp., 1978.

[81] R. W. Taylor and R. L. Frank, "CODASYL data-base management systems," *ACM Computing Surveys*, vol. 8, Mar. 1976.

[82] S. Ceri and J. Widom, "Deriving production rules for constraint maintenance," in *Proceedings of the 16th VLDB Conference* (D. McLeod, R. Sacks-Davis, and H. Schek, eds.), (Brisbane, Australia), pp. 566–577, 1990.

[83] L. G. Bourma and H. Velthuijsen, eds., *Feature Interactions in Telecommunications Systems*. IOS Press, 1994.

[84] T. Bowen, F. Dworack, C. Chow, N. Griffeth, G. Herman, and Y.-J. Lin, "The feature interaction problem in telecommunications systems," in *Seventh International Conference on Software Engineering for Telecommunication Switching Systems (SETTS)*, pp. 59–62, 1989.

[85] G. Kaiser, P. Gross, G. Kc, J. Parekh, and G. Valetto, "An approach to autonomizing legacy systems," in *Workshop on Self-Healing, Adaptive and Self-MANaged Systems*, June 2002.

[86] J. Rumbaugh, "Controlling propagation of operations using attributes on relations," in *Conference proceedings on Object-oriented programming systems, languages and applications*, pp. 285–296, ACM Press, 1988.

[87] A. V. Shah, J. H. Hamel, R. A. Borsari, and J. E. Rumbaugh, "Dsm: an object-relationship modeling language," in *Conference proceedings on Object-oriented programming systems, languages and applications*, pp. 191–202, 1989.

[88] H. J. C. Ellis, S. A. Demurjian, F. J. Maryanski, G. M. Beshers, and J. Peckham, "Extending the behavioral capabilities of the object-oriented paradigm with an active model of propagation," in *Proceedings of the 1990 ACM annual conference on Cooperation*, pp. 319–325, 1990.

[89] DARPA ITO, "Active networks (http://www.darpa.mil/ito/research/anets/)."

[90] G. Su and Y. Yemini, "Virtual Active Networks: towards multi-edged network computing," *Computer Networks*, vol. 36, no. 2/3, pp. 153–168, 2001.

[91] J. Burns, P. Gurung, D. Martin, S. Rajagopalan, P. Rao, D. Rosenbluth, and A. Surendran, "Management of network security policy by self-securing networks," in *DARPA Information Survivability Conference and Exposition (DISCEX II)*, (Anaheim, California), 2001.

[92] L. Ricciulli, "Anetd: Active NETworks Daemon (v1.0)," Tech. Rep. http://www.csl.sri.com/ancors/anetd/, SRI, 1998.

[93] O. Titz, "CIPE - Crypto IP Encapsulation," tech. rep., INKA, May 1997. http://sites.inka.de/bigred/devel/cipe.html.

[94] OMG, "CORBA/IIOP 2.2 specification," Tech. Rep. formal/98-07-01, Object Management Group (OMG), 1998.

[95] E. Decker, P. Langille, A. Rijsinghani, and K. McCloghrie, "Definitions of managed objects for bridges," Tech. Rep. RFC 1493, IETF, July 1993.

[96] ITO, "Active network backbone (ABone)," Tech. Rep. ISI (http://www.isi.edu/abone).

[97] T. Bray, J. Paoli, C. Sperberg-McQueen, and E. Maler, "The eXtensible Markup Language (XML) 1.0," tech. rep., W3C, 2000.