Towards Self-Configuring Networks[*†]

Alexander V. Konstantinou, Danilo Florissi, Yechiam Yemini
DCC Lab, Computer Science Department, Columbia University
{akonstan, df, yemini}@cs.columbia.edu

Abstract

Current networks require ad-hoc operating procedures by expert administrators to handle changes. These configuration management operations are costly and error prone. Active networks[2, 3] involve particularly fast dynamics of change that cannot depend on operators and must be automated. This paper describes an architecture called NESTOR that seeks to replace labor-intensive configuration management with one that is automated and software-intensive. Network element configuration state is represented in a unified object-relationship model. Management is automated via policy rules that control change propagation across model objects. Configuration constraints assure the consistency of model transactions. Model objects are stored in a distributed repository supporting atomicity and recovery of configuration change transactions. Element adapters are responsible for populating the repository with configuration objects, and for pushing committed changes to the underlying network elements. NESTOR has been implemented in two complementary versions and is now being applied to automate several configuration management scenarios of increasing complexity, with encouraging results.

Index Terms -- configuration management, network modeling, change propagation, polic, self-organizing systems, active networks, directory services.

1 INTRODUCTION

Current networks require ad-hoc operating procedures by expert administrators to handle changes -- from installing or removing network elements, to reconfiguring them. These configuration change management operations are costly, error prone, can result in unpredictable failures and inefficiencies, may involve costly recovery and limit the speed of network change dynamics. Active networks[2, 3] involve particularly fast dynamics of changing element configurations due to the downloading and executing of Active Applications (AAs). An AA needs to configure its own parameters, and change those of its Execution Environment (EEs); the EE, in turn, may have to change node configuration parameters. These changes cannot depend on operators and must be automated as part of launching an AA. Furthermore, the software that automates configuration change management may need to be dynamically updated as new AAs are loaded and executed; it is therefore active itself and requires specialized configuration management AAs. In general, a *self-configuring* network is one that automates configuration management. This paper describes self-configuring network technologies developed by the NESTOR project[4].

Several factors make the design of self-configuring networks challenging:

1. **The change propagation problem**: A configuration management task typically requires changes in multiple interdependent elements. For example, provisioning a frame relay virtual circuit to support an IP link between two routers requires configuration changes in underlying multiplexers, frame relay switches and routers. Self-configuring software needs to:
   - Recognize these different elements, their relationships and configuration states – *network topology discovery*;
   - Represent the knowledge of the sequence of changes in these elements – *change propagation rules*;
   - Effect the changes in each element through heterogeneous widely varying proprietary instrumentations, 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*.
2. **The configuration policy problem**: Configuration changes may lead to inconsistent configuration states resulting in operational failures and inefficiencies. For

[*] Research sponsored by DARPA contract DABT63-96-C-0088
[†] Parts of this paper have been previously published in [1]

example, consider an active network AA to prevent denial-of-service (DOS) attack through traffic filtering. This AA requires configuration of the respective EE, the node OS and the network hardware classifiers. A mismatched configuration could lead to inefficient allocation of underlying resources, turning the active node into a traffic bottleneck, potentially increasing the damage of a DOS attack. An inconsistent configuration may cause not only traffic loss, but also intermittent crash of the node. Therefore, a self-configuring network needs to:

- Represent policy knowledge about configuration consistency relationships – *represent policy* constraints
- Enforce these policy constraints to assure consistent configurations – *enforce* policies
- Enable organizations to program policy constraints to effect their operational policies – *programmable 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, when an IP link is provisioned over a frame-relay VC, the change propagation rules and policies associated with underlying multiplexers, frame relay switches and routers must be composed to effect the configuration changes associated with these elements.

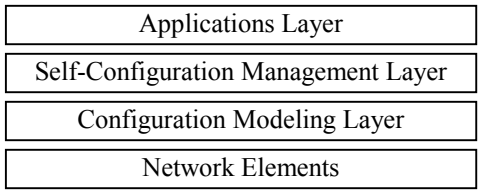

Figure 1: Architecture of a Self-Configuring Network

These problems suggest a four-layered architectural organization of self-configuring networks as depicted in Figure 1. Applications that access or activate self-configuration tasks execute at the top layer. The Self-Configuration Management layer maintains and applies change propagation rules and policy constraints to the Configuration Model. The Configuration Modeling layer consists of software to discover and maintain network topology and element configuration data. The Configuration Modeling layer uses adapters to monitor and control the underlying network elements. At the lowest layer reside the Network Elements to be managed.

In a typical scenario, say provisioning a frame-relay VC between IP routers, a provisioning application activates the Self-Configuration Management layer software. This software computes and affects the respective change propagation rules on the Configuration Model, and enforces policy constraints. The Self-Configuration Management software propagates changes to the Configuration Model of the underlying multiplexers, frame relay switches and routers. The Self-Configuration Management layer performs composition as follows. When an element is discovered, its model is instantiated by the Configuration Modeling Layer, which maintains the topology of its relationships with other elements. The change propagation rules and policy constraints associated with this element are then compiled with change propagation and constraints of related elements. These change propagation rules and policy constraints are then affected whenever a respective change applies to the element. Committed changes to the Configuration Modeling layer are then pushed to the actual network elements through adapters.

For example, consider an active network using an AA for protection against denial-of-service (DOS) attack. When the Configuration Modeling Layer discovers a new source of traffic:

1. The model is updated to indicate the relationship of this new source to the active nodes in the network;
2. The DOS protection application, at the Application layer, may dispatch AAs to several of these active nodes to filter traffic from the new source;
3. This results in updates to the Configuration Model to reflect these new AAs;
4. When these AAs are installed at the respective active nodes, the change propagation rules of the Self-configuration Management layer are activated to configure the EEs and node resources;
5. Similarly, policy constraints are enforced to assure that the AA, EE and node configurations are consistent with operations policies;
6. Once this configuration change transaction is completed, the Configuration Modeling layer affects these changes to network elements and activates the AAs.

Notice that the architectural model enables multiple approaches to organize such active DOS protection. It is possible that the control logic of the protection AAs resides entirely with the active DOS protection applications, and the Self-Configuration layer is only used to propagate configuration changes and enforce operations policies; this is depicted in Figure 2. Alternatively, it is possible that the logic of the active DOS protection entirely resides with the Self-Configuration Management layer. Under this model, the deployment of DOS-protection-AAs is controlled by change propagation rules

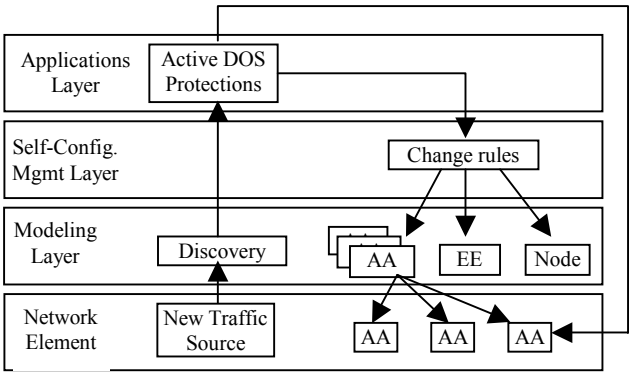

Figure 2: Self-configuring Active Protection Against Denial-of-Service Attacks

and enforced by policy constraints whenever a suspicious new source of traffic is discovered. Other architectures are feasible.

In what follows we proceed with a more detailed discussion of a self-configuring network architecture and respective technologies, developed by the NESTOR system to implement the architectural strategy introduced above.

NESTOR is concerned with several technical challenges:

a. How to unify access to heterogeneous configuration databases and repositories so that configuration management tasks can be programmed and executed by software rather than manually,
b. How to code knowledge of configuration consistency rules in a composable form, and enforce these rules through configuration changes,
c. How to support rollback and/or recovery of operational configuration states,
d. How to detect and handle emergent inconsistencies between configuration states and states controlled by underlying built-in procedures.

NESTOR provides comprehensive self-configuring network software:

- All configuration changes can be systematically automated reducing the complexity, time and cost of operations administration,
- System integrity and configuration consistency through changes can be assured, unlike current manual operations,
- Dynamic changes can be entirely automated; a network can change subject to very rapid dynamics, which is not possible with current manual operations,
- A network can be restored to a consistent configuration and recover from failure modes automatically without complex actions by operations administrators.
- Operations administration becomes scalable, with operations administration staff and expertise leveraged to manage a network of any size through programmed self-configuring configuration management rules and policies.

The rest of the paper is organized as follows. Sections 2 and 3 introduce the role of data and semantic modeling in network management. Section 4 presents the NESTOR, architecture and illustrates its operations through an example. Section 5 discusses the prototype NESTOR implementation, followed by Section 6 on related work, Section 7 on future work. The paper concludes with a summary section.

2 CONFIGURATION DATA MODELING

The goal of configuration modeling is to provide a unified view of all data and knowledge needed to support automated configuration management. Currently, configuration information is spread across different element-specific repositories. Relationships between different configuration elements are implicit and require the development of special tools to be discovered. Gathering, correlating, and visualizing a system-wide picture of configuration is a daunting and sometimes impossible task. Different repositories contain replicated and interdependent configuration information, which can often be inconsistent. Unlike network monitoring, which has benefited from the wide adoption of the Simple Network Management Protocol (SNMP)[5], there has been no widely accepted standard for network configuration. Each repository employs a different and vendor-specific mechanism for accessing and manipulating configuration information. Configuration modeling addresses these issues by providing a unified semantic layer enabling the creation of portable, vendor-independent configuration tools.

Configuration models in the NESTOR system are expressed using 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 well as of distribution. Finally, objects encapsulate the methods for accessing the underlying element instrumentation. Object-based approaches have

been previously applied in modeling networks in various systems including NETMATE[6], SMARTS InCharge[7], CMIP/GDMO[8], Dolphin[9] and DEN[10]/MOF[11].

The current implementation of RDL is a subset of MODEL[12, 13], a language for modeling network systems for event correlation. MODEL extends the CORBA Interface Definition Language (IDL)[14] with support for instrumented and computed attributes, declaration of problems (events), and association relationships for modeling event propagation. Instrumented attributes are bound to values stored in the managed element, whereas computed attributes are bound to an expression that is evaluated dynamically.

```
interface anets::ActiveNode
        : nestor::system::NetworkedSystem {
  readonly attribute String nodeOSVersion;
  relationshipset serves,
                  ExecutionEnvironment,
                  servedBy;
}
interface anets::ExecutionEnvironment
          : nestor::system::Process {
  attribute int anepID "Anet ID";
  relationship servedBy, ActiveNode,
               serves;
  relationshipset serves,
                  ActiveApplication,
                  servedBy;
}
interface anets::ActiveApplication { … }
```

Figure 3: Resource Definition Language Examples

Figure 3 depicts fragments of the model of an Active Network[15] node 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 standards bodies. Interface definitions may include attribute, and relationship declarations. In the `ActiveNode` example, the first statement declares a read-only string attribute named "`nodeOSVersion`", which stores the version of the Active Networks NodeOS[16] specification supported by this Active Node. The second statement declares a to-many association between this interface and classes implementing the interface `ExecutionEnvironment`. Associations are declared by naming both ends (role names), the type of the association class, and the multiplicity of the association (to-one, to-set, or to-sequence). In the example, the association between `ActiveNode` and `Execution Environment` is specified as one-to-many. The model reflects the fact that objects of type `ActiveNode` may host one or more Active Execution Environments (EEs). The relationship `servedBy` goes in the other direction, from an `ExecutionEnvironment` to an `Active Node`. The "`nestor::system`" scope in the declaration of `ActiveNode` denotes the NETMATE[6] schema, which serves as the base classes for the construction of NESTOR classes.

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 `ActiveNode` interface inherited from `NetworkedSystem` (omitted for brevity). 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.

3 CONFIGURATION SEMANTIC MODELING

While object models capture structure and relationships, via inheritance and associations, they do not make any statements on the values of the modeled objects. For example, the `anepID[17]` attribute definition in `Execution Environment` does not state any restrictions on the value of the attribute in one instance in relation to other instances. Similarly, an Active Application may need to configure itself to transmit IP datagrams that do not exceed the maximum transfer unit of the local link-layer interface. In the NESTOR system, such restrictions and relations are respectively expressed as constraints and propagation rules on the values of one or more objects.

Constraints on configuration objects and relationships enrich the model, and can be used to automate detection and reaction to inconsistencies. For example, constraints may express that a specific `anepID` has been reserved to a specific EE as identified by the signature on the EE executable. An attempt to deploy an EE registering the same `anepID` that has not been signed by the matching principal would be rejected and rolled-back.

The *Constraint Definition Language* (CDL) is a declarative expression language for stating assertions over the valid values of objects in RDL. As an inherent language feature, statements in CDL cannot modify any attributes or relationships in the model and do not cause

side effects. Constraints may be composed from restrictions on the configuration of component devices or services. E.g., "all user home directories must be backed up". This statement applies to two services that are usually separate, a network information service for user accounts, and the configuration of network backup services. Another example is "the IP interface configuration of every node connected to a switch must match the VLAN configuration active on its port".

The current implementation of CDL is based on the Object Constraint Language (OCL)[14]. OCL was developed as part of the Unified Modeling Language (UML) standard in order to formally define the semantics of the UML. Unlike OCL statements, CDL separates the object model from the constraint definitions for two reasons. First, the most interesting constraints are the ones that make statements about the configuration of multiple RDL interfaces. In such cases, it may not be clear which object should "own" the constraint. For example, the aforementioned backup constraint is as much a property of the user account as of the backup service. Second, the same manager will not always perform model authoring and constraint authoring. Device and service models will usually be obtained from the vendor, or may be bundled in some standard model package. Attaching domain-specific constraints to RDL interfaces will limit the sharing of these models.

```
nestor::system::NetworkedSystem::->allInstances
  ->select(h | h.hostname <> null)
  ->forAll(h1, h2 | h1 <> h2 implies
           h1.hostname <> h2.hostname);
```

Figure 4: Constraint Definition Language Example

A simple CDL constraint is shown in Figure 4. The constraint states that for all object instances implementing the RDL interface `nestor::system::NetworkedSystem`, those who have a non-null name should all have different names. In the OCL syntax, the right arrow operator (`->`) operates on collections of objects (sets, bags, and sequences). The `allInstances` operator collects over all classes implementing a particular interface. Select is an operator that filters out elements in a collection that do not satisfy the Boolean expression condition. In this case, select will remove all IP hosts that have a null name. Finally, the `forAll` operator states that for every pair of IP host instances, the following Boolean expression on the remaining IP host instances must be valid: "if two host objects are different (different instances), then their names must be different".

The *Policy Definition Language* (PDL) is used to assign values to configuration model objects based on the configuration of related objects. For example, the maximum datagram size of an Active Application may be set to the minimum MTU of the Active Node's link-layer interfaces. In the NESTOR system, such dependencies are expressed as acyclic spreadsheet-style change propagation rules. PDL rules use the same OCL syntax used by CPL constraints, only the result does not have to be a Boolean value and must be assigned to an attribute of the object instance selected. A PDL rule example is shown below in Figure 5. This rule states that the `maxDatagramBytes` configuration attribute of a particular Active Application must be set to the minimum of the Active Node's non-local (excludes loop-back) interfaces. Note the navigation from the AA to the enclosing EE using the `servedBy` relation and then the enclosing Active Node to obtain the list of link interfaces. More complex policies can be expressed. For example, if the AA has a concept of a peer, its `maxDatagramBytes` attribute may be set to the minimum MTU on the path to the peer. It is possible to discover this minimum MTU value by navigating the relations in the unified configuration model.

```
columbia::MyActiveApplication->allInstances
  ->assign(a | a.maxDatagramBytes,
     min(a.servedBy.servedBy.linkInterfaces
         ->select(i |
                  not i.isLoopBack).mtu))
```

Figure 5: Policy Definition Language (PDL) Example

4 NESTOR ARCHITECTURE AND OPERATIONS

The overall architecture of the NESTOR system is depicted in Figure 6. In the top layer, self-configuring *Applications* access a unified semantic configuration model to discover the configuration of their environment and to export their own configuration state, operational constraints, and change propagation rules. Examples of such applications include Active Networks applications and Execution Environments, network and systems management utilities, intrusion and denial of service detection applications, as well as topology aware applications such as peer-to-peer applications. Systems administrators may interactively access the configuration repository through graphical or text-based user interface tools, or they may execute scripts or programs tailored specifically for a particular task. NESTOR Applications access the repository using the *Directory Access Protocol* (DAP), a remote interface permitting applications to execute either locally or remotely.

NESTOR uses *protocol proxies* to interface with legacy *dynamic* configuration protocols. Existing configuration servers, such as Dynamic Host Configuration Protocol (DHCP)[18] servers, are replaced by NESTOR protocol proxies. Clients connecting to the proxy server continue to receive the same service with the difference that changes are effected through the NESTOR repository. In the DHCP example, it is the NESTOR DHCP proxy server which picks up the host discover request. The proxy server then initiates a repository transaction, looks up the `IpLeasing` instance responsible for the host's network and invokes the lease method. Depending on the implementation of the `IpLeasing` object, the request may be handled by contacting a real DHCP server, or by implementing the IP leasing policy internally. Before returning the IP leased address (if one was found) the `IpLeasing` object will update its map of unique client identifiers (commonly Ethernet hardware addresses) to addresses. Once the lease method returns, the DHCP proxy will attempt to commit the update transaction. If no constraints have been violated, the transaction will be successfully committed, and the DHCP proxy will return the leased address to the requesting host. In most cases, however, the configuration changes effected by the `IpLeasing` service will need to be propagated within the repository. This can be by adding a propagation rule from the current address of an IP interface and its permanent name into the model of the matching DNS address record.

The *Directory Management Protocol* (DMP) is used between NESTOR Resource Directory Servers to support distribution, replication, and caching of resource objects. Similarly to directory services, NESTOR offers mission-critical services that must be available even in the face of server or network failures. Distribution of NESTOR services is also important for several reasons. (1) Although similar repositories[7] used in event correlation have been shown to scale well (to the order of hundreds of thousands of objects), there is ultimately a limit to the number of modeled objects that can be stored and maintained in a single server. (2) The wide geographical dispersion of some networks requires distribution for timely response. (3) Finally, the breakdown of administrative domains often forces the distribution of services that may not be technically required otherwise.

The *Self-Configuration Management layer* consists of a constraint and change propagation manager responsible for authorizing changes in the model, maintaining consistency through change propagation, and assuring that the composition of the change propagation rules does not lead to cyclical changes. The constraint and propagation manager subscribes for changes in the model and has the right to abort configuration transactions, or to effect additional changes. Its actions are controlled by CPL constraints and PDL rules. Applications may install additional constraints and propagation rules; however, additional rules may not create cycles in attribute dependencies. The manager assures network configuration consistency because configuration model changes are not applied to the real world configuration repositories unless all affected propagation rules have been evaluated and all constraints are satisfied.

The *Configuration Modeling layer* is responsible for maintaining the model and supporting the advanced model operations. The *Resource Directory Server* (RDS) maintains an object *repository* that stores and controls

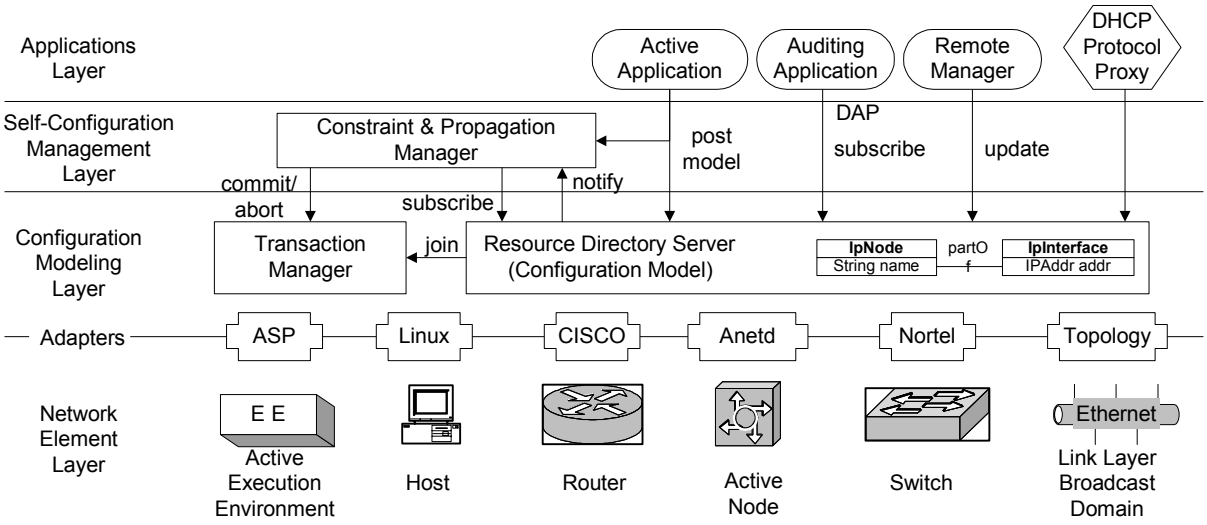

Figure 6: NESTOR Architecture

access to model object instances. Repository objects reflect configuration settings at the real network elements plus meta-information that is supplied or inferred from multiple sources. For example, a model object representing a network host may contain information instrumented from the host, such as network interface configuration, meta-information such as host ownership, and values such as the host's name which are replicated in various repositories. The DAP interface provides operations for creating, committing, and aborting transactions, supports object queries, as well as operations for creating, updating, and deleting objects.

The *Protocol Adapter Layer* provides instrumentation for network elements that are not NESTOR-enabled. *Adapters* are responsible for propagating information, forward and backward, between the RDS repository and the managed element or service. Use of protocol adapters separates the task of mapping the unified model attributes to the real element attributes, from the protocols realizing that mapping. For example, the SNMP[5] adapter may be used to simplify the implementation of an object supporting the `NetworkedSystem` interface. Access requests to host attributes can be translated into SNMP GET/SET operations. The host's name attribute, for instance, may be mapped to the SNMP `System.sysName` object. Multiple adapters may be used in instrumenting the attributes and methods of a particular object. Unfortunately, in some cases, especially at the network layer, protocols do not support remote access to all configuration parameters. For example, the original DNS protocol did not provide an update operation and relied on an implementation specific configuration (usually performed by editing the zone file). In such cases, adapters have to be customized for the particular service implementation, taking into account the host operating system and particular service version. NESTOR supports adapters of the following standard protocols: SNMP, DNS, DHCP, LDAP[19], NIS/NIS+[20], NDS[21] and Windows NT Active Directory[22].

4.1 EXAMPLE: ANETD MANAGEMENT

The operations of NESTOR will be illustrated through an example of managing the Active Networks Daemon (Anetd)[23]. Anetd is currently being used to deploy and manage EEs on the DARPA ABONE[24].

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 Figure 3. In this example, fragments of two interfaces are shown in Figure 7, one for modeling an Anetd process (`Anetd`), and another for the EEs hosted (`AnetdProcess`).

```
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, …
}
```

Figure 7: Anetd daemon and process RDL definitions

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).

```
anets::ActiveNode->allInstances
  ->collect(an | an.servesApplications)
  ->select(app: System::Application |
           app.oclIsKindOf(anetd::Anetd))
  ->select(ad : anetd::Anetd | ad.isPrimary)
  ->size = 1
```

Figure 8: Exactly one primary Anetd per Active Node (CDL constraint)

Once the RDL data model has been designed, the model author may add intrinsic constraints and propagation rules expressed in the *Constraint Definition Language* (CDL) and the *Propagation Definition Language* (PDL). For example, it may be stated that there should be exactly one primary `Anetd` within each `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 CDL example for the former constraint is shown in Figure 8. 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 Figure 9 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 iteratively assigning true to the first object's isPrimary attribute, and false to the others.

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 Figure 10 sorts the list of Anetd objects in reverse and sets the childPort attribute of each object to that of the previous one (the tail is assigned port 0). 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.

```
anets::ActiveNode->allInstances
  ->collect(an | an.servesApplications)
  ->select(app |
           app.oclIsKindOf(anetd::Anetd))
  ->sortBy(ad: anetd::Anetd |
           ad.port, int.less-than)
  ->iterate(ad: anetd:Anetd;
            count: int = 0 |
    if (count = 0)
      assign(ad.isPrimary, true,
             count + 1)
    else
      assign(ad.isPrimary, false,
             count + 1)
    endif
```

Figure 9: Primary Election (PDL propagation rules)

```
anets::ActiveNode->allInstances
  ->collect(an | an.servesApplications)
  ->select(app |
           app.oclIsKindOf(anetd::Anetd))
  ->sortBy(ad | ad.port, int.greater-than)
  ->iterate(ad : anetd::Anetd;
            port : int = 0 |
    assign(ad.childPort, port, ad.port))
```

Figure 10: Forwarding chain (PDL propagation rules)

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[17] 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[25] 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 11.

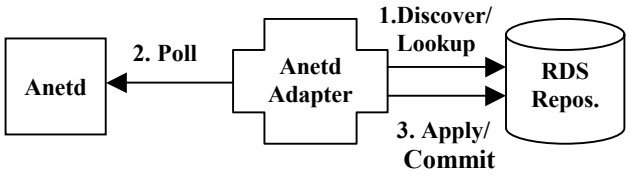

Figure 11: Anetd Read Instrumentation

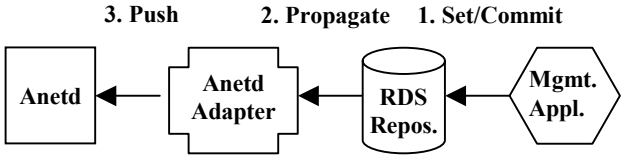

Figure 12: Anetd Set Instrumentation

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 12. 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, 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 Figure 9 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[26]) 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.

5 IMPLEMENTATION

An initial prototype of the NESTOR system was built using the MODEL language and InCharge repository provided by SMARTS[13]. 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, as reported in [1] .

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 found 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[25]. 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, RDS invokes the constraint and propagation manager with a reference to the transaction log. The manager analyzes the log and determines the PDL 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 policy scripts, 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 OCL 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 OCL interpreter that evaluates each expression when detecting a possible violation. Future versions will explore the performance gains of compiling OCL 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[27] (VAN), and Anetd. 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[28] 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 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 13. On the left panel, the browser displays a tree whose first level are repositories, the second level includes the list of available MODEL 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

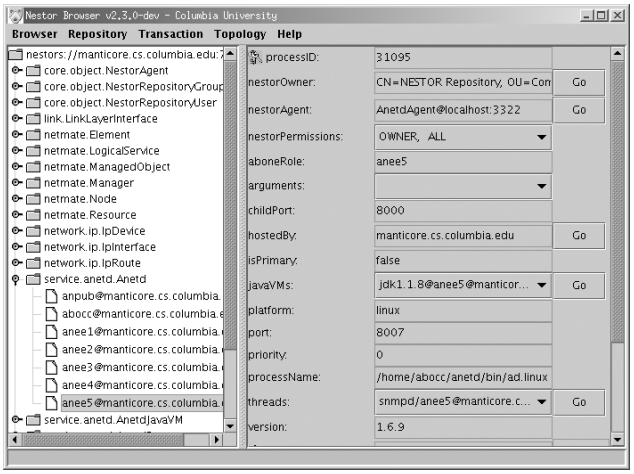

Figure 13: NESTOR Browser

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`". For example, by selecting one of the Anetd "`threads`" relation members (EEs) one can access the EE object configuration. The browser can switch to an update transaction mode to allow for changes. The browser can be executed as a Java WebStart[29] service directly from the repository HTTP server.

The Java NESTOR prototype was first applied to the management of security in dynamic networks, as described in [30]. It has also been used by researchers at Telcordia Technologies as a platform for developing a distributed firewall based on security policies[31]. Currently, the NESTOR prototype is being deployed on the Active Networks Backbone (ABONE) network for instrumentation of Active Nodes, EEs and AAs.

6 RELATED WORK

Other projects in the area of Active Networks management include the SENCOMM[32] project, the ABLE[33] engine, and the ANCORS[34] project.

The SENCOMM project builds on the Smart Packets[35] research that developed a safe language and environment supporting network management functions. In SENCOMM, Smart Probes containing immutable program code, and mutable probe data are transmitted across the network and executed in SENCOMM Management EEs. Probes may collect data across their path, and/or may continue functioning in the management EE. Management EE functions may be extended through the deployment of loadable libraries.

The ABLE research aims to provide AN-based management supporting deployment of distributed IP network management applications. Agents are dispatched to Active Engines that operate as management EEs and are associated with sessions. Agents are executed in a safe (isolated) and secure (authenticated) environment. Management functions are exported through narrow APIs providing access to SNMP MIBS. ABLE extends previous research[36] with packet-based agent delivery, and packet path-based discovery.

Both the SENCOMM and ABLE projects focus on deployment of management agents that collect local performance and configuration information, and may only effect global change through agent-specific protocols. In contrast, the NESTOR approach is focused on enabling and controlling the interaction of agents, rather than providing pure isolation. NESTOR supports navigation of relations across managed systems in a unified model, expression of inter-agent constraints, and propagation rules, as well as, semantic-based fine-grained access control. NESTOR supports both AN-based management, through the deployment of model objects and semantic constraints and propagation rules, as well as management of ANs by allowing AAs and EEs to export their configuration data and semantic models.

The ANCORS project merges technologies from network management and distributed simulation to provide support for runtime assessment of network protocols and operations. ANCORS addresses the integration issues of EEs and AAs by supporting simulation of their operations so that their operations can be closely monitored, and evaluated prior to actual deployment. This is a black-box approach to exposing dependencies between services, and preventing potential resource conflicts. The advantage of this approach is that services need not declare their integrity constraints and propagation rules. The main disadvantage is that simulation can only provide proof of failure, but its results cannot be readily used to support conflict resolution. In such cases, the offending service must either be terminated, or migrated somewhere in the network. NESTOR requires that services expose their configurable properties and semantic constraints and propagation rules. This permits both conflict detection and prevention (without simulation) as well as dynamic conflict resolution. It may be desirable to combine the two approaches, especially in cases where the service models have not been tested out, or are partially incomplete.

NESTOR 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[6, 37-39] and some commercial products (SMARTS InCharge, HP OpenView, Tivoli TME) have used OO resource models successfully to simplify the development of management applications. NESTOR broadens this effort to build on modeling technologies that can create unifying heterogeneous configuration information directory structures to support automated management. The semantic model[38] in NESTOR captures detailed configuration needed to build self-management/organization software.

NESTOR also profits from the large body of 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 information model, initially pursued by the ad-hoc group on Directory Enabled Networks (DEN)[10] and more recently by the DMTF (Distributed Management Task Force)[40]. Future NESTOR versions will support Meta Object Facility (MOF)[11] import/export functions, to assist in leveraging the standards work.

The most closely related management architecture to NESTOR is the ICON system[41, 42] 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)[43]. NESTOR extends these approaches to incorporate multi-protocol access to heterogeneous resource information, configuration transactions, declarative constraints, and constraint propagation through policy scripts.

The Dolphin project[9] 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. NESTOR strives to prevent such failures by checking constraint violations before they adversely affect the consistent configuration of the system.

In the area of configuration management automation, the GeNUAdmin[44] 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[45] 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. NESTOR can support this functionality given an appropriate set of constraints on the unified configuration model. Ganymede[46] 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[47, 48]. Previous work on constraint-based management has been pursued[49, 50]. The focus of these projects has been on employing constraints for the diagnosis of network faults and on algorithms for constraint satisfaction. NESTOR could benefit from these technologies to build its policy script-based propagation of configuration changes across network elements. The NESTOR architecture does not include a specific CSP solver as a core component, but it supports programmatic interfaces for policy scripts to employ their own solvers (including CSP).

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 NESTOR architecture supports the safe use of scripts through the binding of the DAP to libraries for popular interpreted languages, such as Perl[51].

7 FUTURE WORK

Future NESTOR research will focus on discovery, model composition, rule and constraint distribution, as well as maintenance of the mapping between the model and the real world.

It is envisioned that the NESTOR 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. The mechanisms for merging models and performing model-based discovery are currently under investigation.

The current NESTOR prototype supports a distributed repository with centralized constraint verification and change propagation. Mechanisms for distributing these functions are currently under investigation. Finally, the mapping of the real world to the model is a difficult problem of great practical significance. Besides the practical problems imposed by the non-transactional management APIs currently in use (SNMP, Telnet, LDAP), there are fundamental issues that need to be addressed. Unlike databases, changes in network elements cannot always be locked-out. For example, an Ethernet link cannot be guaranteed since it depends on a physical service. Different levels of contracts between network services and the model are being investigated.

8 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 active reconfiguration and programmability for service deployment. The NESTOR system addresses these needs by combining several techniques from object modeling, constraint systems, active databases, and distributed systems in novel management architecture.

In the NESTOR system, management applications operate on a unified object-relationship model of the network using a rich set of operations that support rollback and/or recovery of operational configuration states. Declarative constraints prevent known configuration inconsistencies and in conjunction with policy rules may automatically propagate changes to maintain consistency. Protocol proxies are used to provide much of this functionality with little or no changes in the network clients. A protocol for replication and distribution of the directory assures availability and operational efficiency. NESTOR has been implemented in two complementary versions and is now being applied to automate several configuration management scenarios of increasing complexity, with encouraging results.

REFERENCES

[1] Y. Yemini, A. Konstantinou, and D. Florissi, "NESTOR: An Architecture for Self-Management and Organization," *IEEE JSAC*, vol. 18, 2000.

[2] D. L. Tennenhouse and D. J. Wetherall, "Towards an Active Network Architecture," *Computer Communication Review*, vol. 26, 1996.

[3] DARPA ITO, "Active Networks (http://www.darpa.mil/ito/research/anets/)."

[4] DCC Laboratory Columbia University, Y. Yemini, A. Konstantinou, and D. Florissi, "NESTOR (http://www.cs.columbia.edu/dcc/nestor)."

[5] J. Case, M. Fedor, M. Schoffstall, and J. Davin, "A Simple Network Management Protocol," IETF RFC 1067, 1988.

[6] 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.

[7] System Management Arts (SMARTS), "InCharge," 1.5.2 ed. White Plains, NY, 1997.

[8] ITU-T, "Information technology - open systems interconnection - structure of management information : guidelines for the definitions of managed objects," Recommendation X.722, ISO/IEC 10165-4, 1992.

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

[10] S. Judd and J. Strassner, "Directory-enabled Networks : Information Model and Base Schema," DEN Ad Hoc Working Group Version 2.0.2-2, 1998.

[11] OMG, "Meta Object Facility (MOF) Specification," Version 1.3, 1999.

[12] D. Ohsie, A. Mayer, S. Kliger, and S. Yemini, "Event Modeling with the MODEL Language : A Tutorial Introduction," SMARTS (System Management Arts), 14 Mamaroneck Ave., White Plains, New York, 10601, White Plains.

[13] System Management Arts, "MODEL Language Reference Manual," White Plains, NY 1996.

[14] OMG, "Object Constraint Language Specification," Object Management Group (OMG) ad/97-08-08 (version 1.1), September 1, 1997 1997.

[15] K. L. Calvert, "Architectural Framework for Active Networks (Version 1.0)," University of Kentucky July 27, 1999 1999.

[16] L. Peterson and Active Networks Node OS Working Group, "NodeOS Interface Specification," Princeton University January 10, 2001 2001.
[17] S. Dawson, F. Gilham, M. Molteni, L. Ricciulli, and S. Tsui, "User Guide to Anetd 1.6.9," SRI (http://www.csl.sri.com/ancors/anetd/), 2001.
[18] R. Droms, "Dynamic Host Configuration Protocol," IETF RFC 1531, 1993.
[19] W. Yeong, T. Howes, and S. Kille, "Lightweight Directory Access Protocol," IETF RFC 1777, March 1995 1995.
[20] Sun Microsystems, "NIS : Network Information Service."
[21] Novell Inc., "Netware Directory Services."
[22] Microsoft Corp., "ADSI : Active Directory Services Interfaces." Redmond, WA.
[23] L. Ricciulli, "Anetd: Active NETworks Daemon (v1.0)," SRI http://www.csl.sri.com/ancors/anetd/, 1998.
[24] DARPA ITO, "Active Network Backbone (ABone)," ISI (http://www.isi.edu/abone).
[25] Sun Microsystems, "Jini Architecture Specification," Palo Alto, CA 1998.
[26] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler, "The Extensible Markup Language (XML) 1.0," W3C Recommendation 2000.
[27] G. Su and Y. Yemini, "Virtual Active Network (VAN)," DCC Laboratory, Columbia University (http://www.cs.columbia.edu/dcc/van/), 2000.
[28] T. Dierk and C. Allen, "The TLS Protocol version 1.0," IETF RFC 2246, 1993.
[29] Sun Microsystems, "Java WebStart," 1.0.1 ed: http://java.sun.com/products/javawebstart, 2001.
[30] A. V. Konstantinou, Y. Yemini, S. Bhatt, and S. Rajagopalan, "Managing Security in Dynamic Networks," presented at 13th USENIX Systems Administration Conference (LISA'99), Seattle, WA, USA, 1999.
[31] J. Burns, P. Gurung, D. Martin, S. Rajagopalan, P. Rao, D. Rosenbluth, and A. V. Surendran, "Management of Network Security Policy by Self-securing Networks," presented at DARPA Information Survivability Conference and Exposition (DISCEX II), Anaheim, California, 2001.
[32] A. W. Jackson, J. P. G. Sterbenz, M. N. Condell, and R. R. Hain, "Active Network Monitoring and Control: The SENCOMM Architecture and Implementation," presented at DARPA Active Networks Conference and Exposition (DANCE), California, 2002.
[33] D. Raz and Y. Shavitt, "An Active Network Approach for Efficient Network Management," presented at IWAN, Berlin, Germany, 1999.
[34] L. Ricciulli, P. Porras, P. Lincoln, P. Kakkar, and S. Dawson, "An Adaptable Network COntrol and Reporting System (ANCORS)," presented at DARPA Active Networks Conference and Exposition (DANCE), California, USA, 2002.
[35] B. Schwartz, A. W. Jackson, W. T. Strayer, W. Zhou, R. D. Rockwell, and C. Partridge, "Smart Packets: applying active networks to network management," *ACM TOCS*, vol. 18, 2000.
[36] Y. Yemini, G. Goldszmidt, and S. Yemini, "Network Management by Delegation," presented at Second IFIP/IEEE International Symposium on Integrated Network Management, Washington, D.C., 1991.
[37] 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.
[38] Y. Yemini, A. Dupuy, S. Kliger, and S. Yemini, "Semantic Modeling of Managed Information," presented at Second IEEE Workshop on Network Management and Control, Tarrytown, NY, 1993.
[39] F. Teraoka, Y. Yakote, and M. Tokoro, "A Network Architecture Providing Host Migration Transparency," *Computer Communication Review*, vol. 23, 1991.
[40] Distributed Management Task Force (DMTF), "Common Information Model (CIM) Specification," Version 2.2, June 14, 1999 1999.
[41] S. K. Goli, J. Haritsa, and N. Roussopoulos, "ICON: A System for Implementing Constraints in Object-based Networks," presented at Integrated Network Management, IV, 1995.
[42] J. Haritsa, M. Ball, N. Roussopoulos, A. Datta, and J. Baras, "MANDATE: MAnaging Networks using DAtabase TEchnology," *IEEE Journal on Selected Areas in Communications*, vol. 11, pp. 1360-1371, 1993.
[43] J. e. Widom and S. e. Ceri, *Active database systems: triggers and rules for advanced processing*. San Francisco, CA: Morgan Kaufmann, 1996.
[44] M. Harlander, "Central system administration in a heterogeneous unix environment," presented at 8th USENIX System Administration Conference (Lisa VIII), 1994.
[45] J. Finke, "Automation of site configuration management," presented at 11th USENIX System Administration Conference (Lisa '97), 1997.
[46] J. Abbey and M. Mulvaney, "Ganymede: An Extensible and Customizable Directory Management Framework," presented at LISA XII, Boston, MA, 1998.
[47] E. C. e. Freuder and A. K. e. Mackworth, *Constraint-based reasoning*: MIT Press, 1994.
[48] E. Tsang, *Foundations of Constraint Satisfaction*: Academic Press - Harcourt Brace & Company, 1993.
[49] M. Sabin, R. D. Russel, and E. C. Freuder, "Generating Diagnostic Tools for Network Fault Management," presented at The Fifth IFIP/IEEE International Symposium on Integrated Network Management (IM'97), San Diego, CA, 1997.
[50] M. Sabin, A. Bakman, E. C. Freuder, and R. D. Russel, "Constraint-Based Approach to Fault Management for Groupware Services," presented at International Symposium on Integrated Network Management (IM'99), Boston, MA, 1999.
[51] L. Wall, T. Christiansen, R. Schwartz, and S. Potter, *Programming Perl*, 2 ed: O'Reilly & Associates, 1996.