Verification of Network Management System Configurations

David L. Cohrs
(608) 262-6617 dave@cs.wisc.edu
Barton P. Miller
(608) 262-3378 bart@cs.wisc.edu

Computer Sciences Department
University of Wisconsin − Madison
1210 W. Dayton Street
Madison, Wisconsin 53706

ABSTRACT

The size and complexity of current computer internets are increasing the need for automated network management. In the past, networks were usually managed from a central location. Today, internets are too large, and individual administrative domains too autonomous for this centralized approach. Distributing the network management system causes problems, because there is no longer central control over the configuration of the parts of the system. We address these problem through the use of a high level, formal specification language, NMSL. NMSL allows the network administrators to describe their network environment and its relationship to other environments. The NMSL system then operates in two roles: it verifies network management specifications, and it automatically configures network management systems given a verified specification.

This paper presents a model for network management systems, and a method for verifying specifications of these systems. We divide the verification problem into three parts: capacity, protection, and configuration verification. Capacity verification determines if the processes in the network management system are configured to handle the load that their clients place on them. Protection verification determines if access permissions are being violated. Configuration verification determines if other general requirements on the specification are being met. We also provide a way to distribute the verification process, and a way to summarize information that needs to be propagated across domain boundaries. We discuss the performance of our implementation of this system, and describe our future research directions.

Keywords: Management Languages, Inter-Organizational Management Issues

---

Research supported in part by an AT&T Ph.D Scholarship, NFS grant CCR-8815928, ONR grant N00014-89-J-1222, and a Digital Equipment Corporation External Research Grant.

1. INTRODUCTION

Computer internets are growing in size and complexity, and the need for automated network management systems is growing as well. As the size of a network management system increases, it also grows in complexity. If the network is small, the task of network management can be performed by a human administrator, with the use of simple, ad hoc tools. However in a large network, managing the network management system becomes a significant problem in itself. The NMSL system addresses the complexity of managing larger internets by providing a language to specify the configuration of network management systems, and a way to configure network management systems from a specification.

The increased complexity has two main sources. First, a large network management system is similar to any other networking application. It instantiates processes throughout the network. The processes communicate via network management protocols, which must be configured correctly. These processes must be configured to perform the correct internal operations to make queries at the correct time, to answer queries correctly, or to reject invalid queries. Second, these processes are divided along the lines of *administrative domains*. Domains reduce the amount of sharing and coordination possible in configuring the network management system. Each administrative domain needs to remain autonomous − administrators and the owners of a domain are generally not willing to give up control over the management of their networks. However, because management processes need to communicate between these domains, a mechanism is necessary to help coordinate this communication.

The NMSL system provides a solution to the problems involved with network management configuration. The goal of NMSL is to reduce the errors present in the network management system itself, especially those due to incorrect configuration. We employ a high level specification language to achieve this goal. A specification of a network management system can be verified, and a verified specification can be used to directly configure the processes and data bases used by the network management system. The verification process is distributed along domain boundaries, allowing administrative domains to hide any information about themselves that do not pertain to the relationships between domains. An overview of the NMSL system can be found below, with additional information in [5].

NMSL has applications to managing other large, distributed systems as well. A network management system can be thought of as a specialized distributed system. Network management systems include specialized protocols and data models, tailored to the management problem. However, the general structure of NMSL and the NMSL compiler makes the ideas applicable to other data models and protocols.

This paper presents our verification model and the performance of our current implementation of the NMSL verifier. Section 1.1 gives an overview of the entire NMSL system. Section 2 describes the verification model. The verification model includes a formal definition of what it means to verify a network management specification, the method used to verify a specification, the model's limitations, and the effect administrative domains have on the verification process. Section 3 shows an example of how one would apply this model to specifying a simple network management system. Later, in Section 4, we discuss the performance of our translator and the verifier. We describe the current status of the system a summary of our research so far and some conclusions in Section 5.

1.1. NMSL Overview

The NMSL system provides system administrators with a language, NMSL, for describing the configuration of their network management system and the networks that they manage. NMSL specifications depend on a model of network management systems structure, shown in Figure 1.1. In this model, a network management system consists of management *processes* that interact via a management protocol. The processes maintain the state of the network management system in a set of *objects*. The objects are the data or management information maintained by the network management system. The processes manage the hardware attached to the network, which we call *systems*. Systems include all types of network-attached hardware, from mainframe computers and workstations to bridges. Systems are grouped into *domains*. Domains define the administrative boundaries in network management. In existing network management systems, administrative domains are set up in a hierarchical manner, so we support this structuring of domains in our model. Domains are also allowed to overlap, allowing systems (and the processes that manage them) to be members of more than one domain. Overlaps are also permitted in existing network management systems, like SNMP.

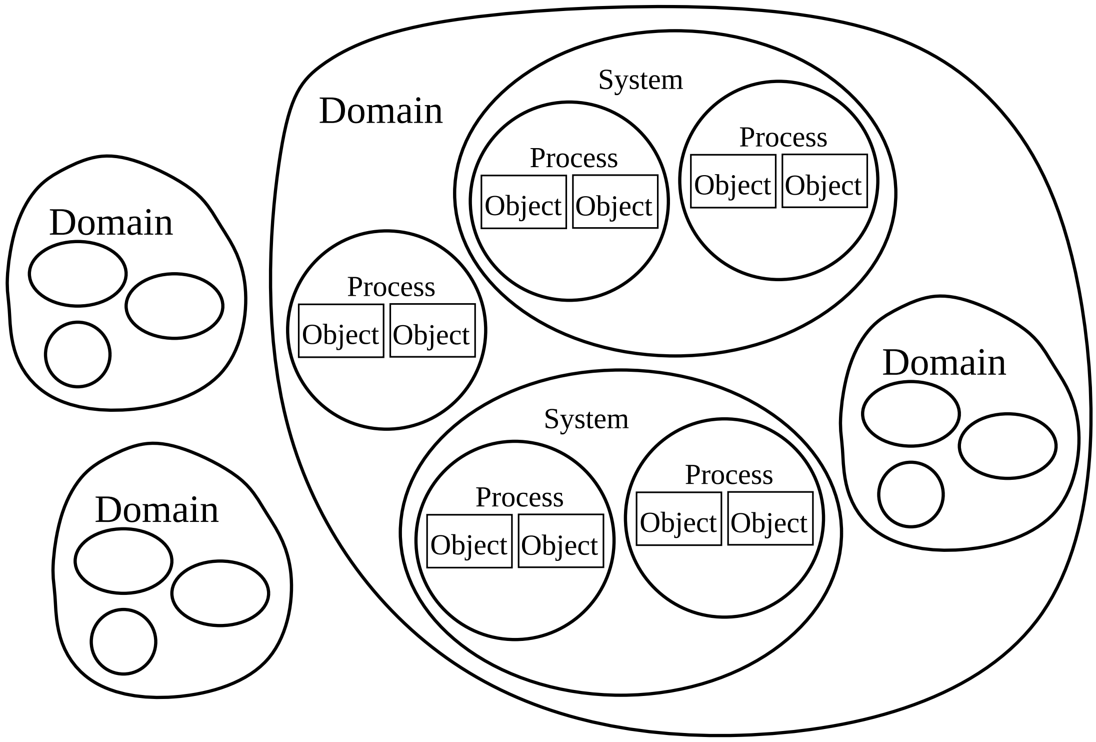

FIGURE 1.1. THE NMSL NETWORK MANAGEMENT SYSTEM MODEL

NMSL allows the specification of objects and processes in terms of abstractions and instantiations. Abstract object specifications define the management information, including structural information (abstract data types), access permissions, naming, and containment. Objects are specified using standard ASN.1 notation[6]. Abstract process specifications describe a process's operation in general terms, including the types and frequencies of queries, the objects the process must be able to access, and the objects the process allows other processes to access. The location or name of the computer on which the process executes is not specified. This models the situation where a common set of processes, along with the objects they manage, will be *instantiated* in a number of locations, but perform the same types of operations in each instantiation. Processes can be parameterized, so that things like the process's peer(s) can be specified when the process is instantiated.

NMSL uses *system* and *domain* specifications to model the physical layout of the network management system. System specifications describe the individual properties of the hardware, e.g. a computer, router or bridge, attached to computer network, and the software that runs on that hardware. They include the instantiations of processes and configuration information, such as the number and types of networks to which the system is connected. Other types of configuration information are the types of management objects defined for the system (for example, EGP objects are only defined for systems which run an EGP router), the operating system, and limitations the hardware may place on the management, such as maximum packet sizes.

Domain specifications describe the administrative grouping of systems and processes. They define a boundary between administrative organizations. Domain specifications list the systems and domains that are members of the domain, and processes that operate on behalf of the domain. Domains can partially overlap as long as the overlap does cause a self-reference.

There are two roles in which NMSL can be used, for describing the network management configuration of an internet, and for configuring the management processes described in the specification. We call these two roles the *descriptive* and *prescriptive* roles of NMSL.

The NMSL compiler is central to both roles of the NMSL system. In its descriptive role, the compiler takes as input the full specification of an administrative domain, and verifies its validity. The exact meaning of validity is the topic of Section 2. Some parts of the specification may include references to the specifications of other administrative domains. In this case, the verifier creates a new specification for this domain, describing only its interface with other domains. Information about the internal structure of a domain is not propagated across domain boundaries. This *external* specification is made visible to other domains, where the verification process takes place again. If at any point an inconsistency is found in a domain's specification, this information must be propagated back the originating domain. If no inconsistencies are found, the specification is deemed to be valid.

Given a valid specification, the NMSL compiler generates prescriptive output in the form of commands to configure the network management processes within the domain. These commands can take many forms, so the NMSL compiler is has the ability to generate many types of output. The systems and processes of the domain are configured using implementation dependent modules, that speak the protocols and have the permission necessary to reconfigure the network management system. Ideally, standard management protocols, such as SNMP[4] or CMIS/CMIP[7, 8] will be used to reconfigure the network management system. Therefore, the NMSL system provides for both the description of the network management system through a specification language, and a method for enforcing that description through automated configuration.

2. THE CONSISTENCY MODEL

The consistency model states the conditions that must be met for a given specification to be correct. It divides these conditions into three categories: *capacity*, *protection*, and *configuration*. These categories correspond to the three types of relationships specified in a NMSL specification. Any given specification, for example, the error messages a router sends and the conditions under which they occur, are defined in terms of these three relationships. The capacity condition states that a service provider must have enough capacity to handle the requests of all of its clients. The protection condition states that a client must be given permission to perform the requests that it makes. The configuration condition states that individual configuration statements in a specification must meet global constraints. If these three conditions are met in a specification, the specification is consistent.

The existence of administrative domains complicates the problem of determining the consistency of a network management specification. The autonomy and privacy of administrative domains does not allow all the information of a domain's specification to be sent to a central location for the consistency check. Copying all of the specifications to a central location is bad for performance as well. To solve these problems, we divide a domain's specification into two logical components, its internal specification and its external specification. The internal specification defines how the parts of the domain, including subdomains, interact with each other. The external specification defines the how this domain interacts with other domains. Dividing the consistency check along domain boundaries allows information hiding, reduces the search space as compared with a single, centralized check, and distributes the work involved.

In sections 2.1 through 2.3, we describe each of the consistency categories, capacity, protection, and configuration. Section 2.4 describes the effect that domains of administration have on this basic model, and how the internal and external specifications are derived.

2.1. Capacity

The goal of the capacity model is to determine, as quickly as possible, if each service provider has the capacity to provide the services needed by its clients. This is a form of the classic capacity planning problem[3].

Capacity planning provides a systematic approach to modeling and predicting the capacity of a system, in our case, a network management system. To form a capacity planning model, one must determine the parameters that characterize the workload a system, and parameters that are required to predict the future performance of the system. This is an application of performance analysis, with an emphasis on the predictive nature of the model. Designing a capacity planning model requires the creation of an initial model for the system workload, validation, and modifying the model if it does not adequately model the capacity of the system.

In our capacity planning model, we wish to obtain a reasonable answer to the capacity question by use of a simple, easy to understand model. This lead us to employ a system of closed-form equations to solve the capacity planning problem. Closed-form equations have two characteristics we find important. First, since the users of this system will not be performance experts, they need a simple, easily understandable model with simple parameters. Second, this model is important because our capacity problem is part of a larger automated consistency proof, which requires a yes or no answer, and also must execute quickly.

The capacity model assumes a *client/server* based system. Each network management process is considered a client or a server† (or both). These processes are instantiated on systems (hardware) throughout the network being managed. The systems of the network are divided into administrative domains, implying that the processes are also divided into administrative domains. Administrative domains are allowed to nest or overlap, but a domain may not contain itself.

Clients interact with servers by means of a *request*, or query, and a server sends back a *response*. We allow clients to have more than one mode of operation, based on the frequency of queries made. For example, an interactive client could have two modes, an inactive mode, where it is waiting for input, and an active mode, where it is interacting with a human and some servers. In the inactive mode, this client will make few, if any, requests of a server, but in the active mode, it will make frequent requests. We assume that servers all operate in a single mode, that of answering queries from any appropriate client. If a process has the characteristics of both a client and a server, we make the simplifying assumption that these operations are independent.

The capacity model employs an independent, discrete distribution of the frequency of interactions, requests and responses, between network management processes. Interactions are measured in queries per second (qps). We assume the messages involved are sent reliably. We also assume that response time is not a factor in determining the frequency of requests.

Given a group of clients, we can determine the aggregate load that the group places on a server. This aggregate load and its distribution are important for determining the probability of overloading a server, and for propagating information between domains. Given the frequency distribution of the clients, we can determine the average and peak load, as well as calculating the discrete aggregate distribution of the group. When determining the consistency of a specification, we can choose which of these three solutions for the capacity problem is appropriate on a individual basis. The method used depends on the constraints of the client processes. Some clients, such as a client making routing decisions, require time critical information to operate correctly. For such a client, the peak load of each of its servers must be satisfied for the specification to be valid. A less critical client can specify the probability that the server may be overloaded. The average load is appropriate for declaring most network management clients to be valid. Such clients are not on a time critical path, but need to have some assurance that their requests will be satisfied eventually.

To keep the model simple and the problem tractable, a number of concepts are not included in our model. The model does not include information concerning the duration of a given mode − behavior is considered on an instantaneous basis. The model also excludes response time. Because we are not modeling the behavior of the entire network, but just the end-to-end behavior of the processes, we do not have enough information to determine response time. Adding response time to the model makes the problem difficult to solve, if solvable at all. Finally, we do not model the fine-grained operation of the message delivery protocols.

2.1.1. Calculating Capacity

We are interested in determining the capacity, in queries per second, of servers, and in determining the load clients will place on each server, *i.e.* whether each server's capacity will be exceeded. The load can be measured several ways. We are interested in the *average utilization*, the average number of queries per second clients place a server, and the *peak utilization*, the maximum number of queries a server can

†Network management standards[4, 9] refer to these processes as *managers* and *agents*, respectively.

receive per second. We are also interested in the probability that a server's capacity will be exceeded. The utilizations depend on the distribution of requests from the clients, which in turn depend on the number of *modes* of operation of each client. If only one client is being considered, we call its distribution a *simple distribution.* A group of clients has an *aggregate distribution.*

We divide the calculations into two cases: a single client querying a single server, and a group of clients querying a single server. The case of a single client simultaneously querying a group of servers (multicast), while an interesting from a reliability point of view[2], is not used in current network management systems.

The single client/single server case is easy to calculate. For example, consider the interactive client and server shown in Figure 2.1; the average utilization is

$$0.80 \times 0 \; qps + 0.20 \times 20 \; qps = 4 \; qps.$$

The peak utilization is 20 *qps*, and the aggregate distribution is the same as the simple distribution. In general, for a single client and a single server, the average rate is just the sum of the rates for each mode. Determining whether this example is consistent is a matter of determining if the server can withstand the load of its client. In the example in Figure 2.1, the server, which can receive 20 *qps*, has the capacity for both the average and the peak rate of requests. Therefore, the probability that the server's capacity will be exceeded is zero.

In the multiple client/single server case, we simply add the loads of the individual clients to get their average and peak request frequency. For example, Figure 2.2 shows three clients $Client_1$, $Client_2$, and $Client_3$. Their average request rate is

$$\begin{aligned} &0.80 \times 0 \; qps + 0.20 \times 20 \; qps + \\ &0.80 \times 0 \; qps + 0.20 \times 10 \; qps + \\ &0.50 \times 5 \; qps + 0.50 \times 20 \; qps = 18.5 \; qps. \end{aligned}$$

In this case, the server, which can handle 20 *qps*, will not be overloaded in the average case. The peak request rate is the sum of the peak rates for each of the clients,

$$20 \; qps + 10 \; qps + 20 \; qps = 50 \; qps.$$

This is greater than the capacity of the server. The distribution of the aggregate load of these three clients has 8 modes, and is determined using simple probabilities. Basically, we take all combinations of the modes of the three clients. For example, to calculate aggregate Mode 1, we take the combination of Mode 1 of $Client_1$, Mode 1 of $Client_2$ and Mode 1 of $Client_3$. The probability of operating in this aggregate mode is

$$0.8 \times 0.8 \times 0.5 = 0.32$$

and the number of queries per second is

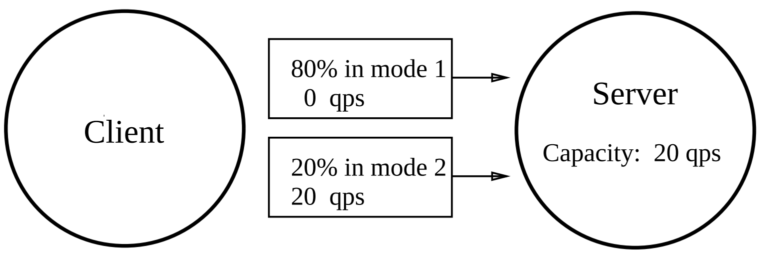

FIGURE 2.1. A SINGLE CLIENT/SINGLE SERVER CONFIGURATION

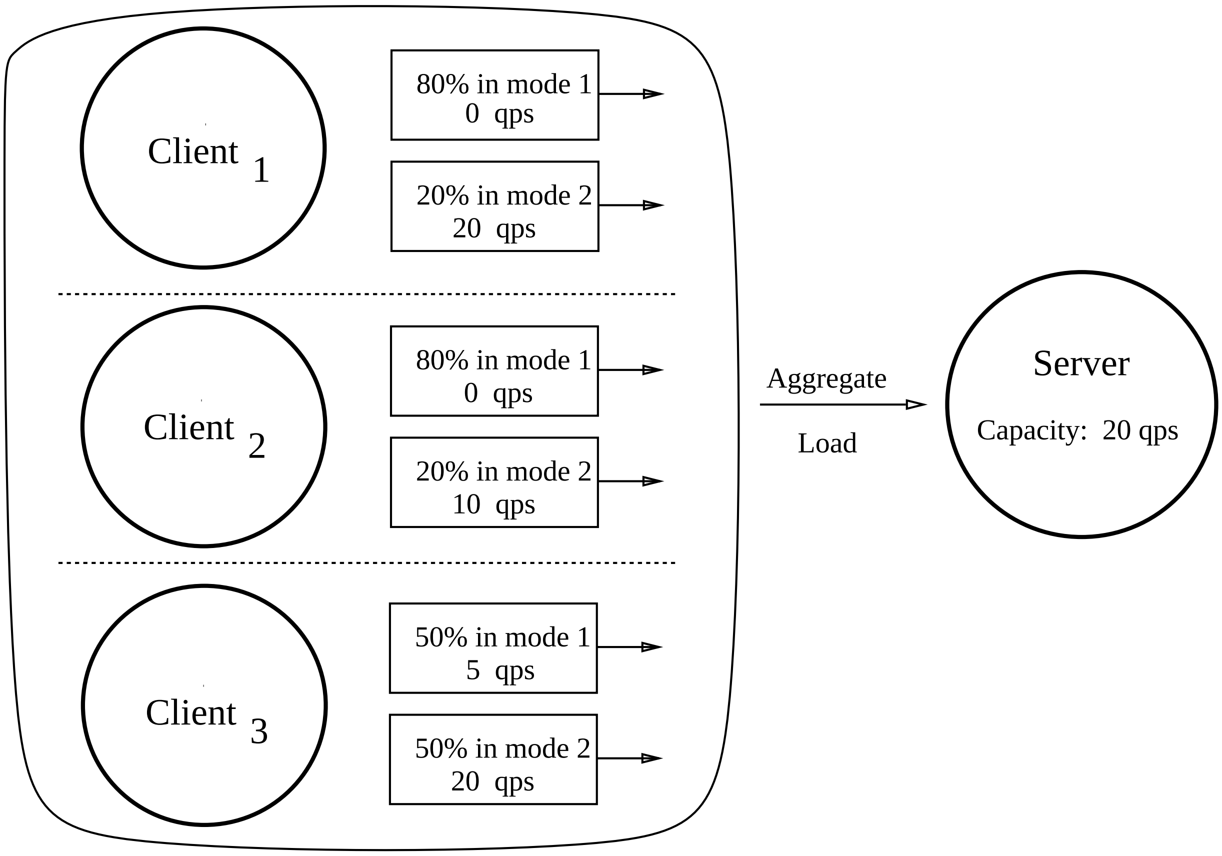

FIGURE 2.2. A MULTI-CLIENT/SINGLE SERVER CONFIGURATION

$$0\ qps + 0\ qps + 5\ qps = 5\ qps.$$

The other modes are calculated in the same manner. The resulting distribution, shown in Figure 2.3, is

Mode 1:	0.32	5 *qps*	Mode 5:	0.08	15 *qps*
Mode 2:	0.32	20 *qps*	Mode 6:	0.08	30 *qps*
Mode 3:	0.08	25 *qps*	Mode 7:	0.02	35 *qps*
Mode 4:	0.08	40 *qps*	Mode 8:	0.02	50 *qps*

From this aggregate distribution, we can see that the probability that the server will be overloaded at any given instant is 0.28 (the sum of the modes with query rates greater than the server's capacity of 20 *qps*, shown as a dotted line in Figure 2.3).

In general, for n clients, $C_1, C_2, \ldots, C_n$, each with m operating modes, or request rates, $F_i(m)$, the probabilities of being in each of the modes, $P_i(m)$, and one server, S, the average aggregate load is

$$\sum_{i=1}^{n} \sum_{j=1}^{m} P_i(j) F_i(j).$$

The maximum load is

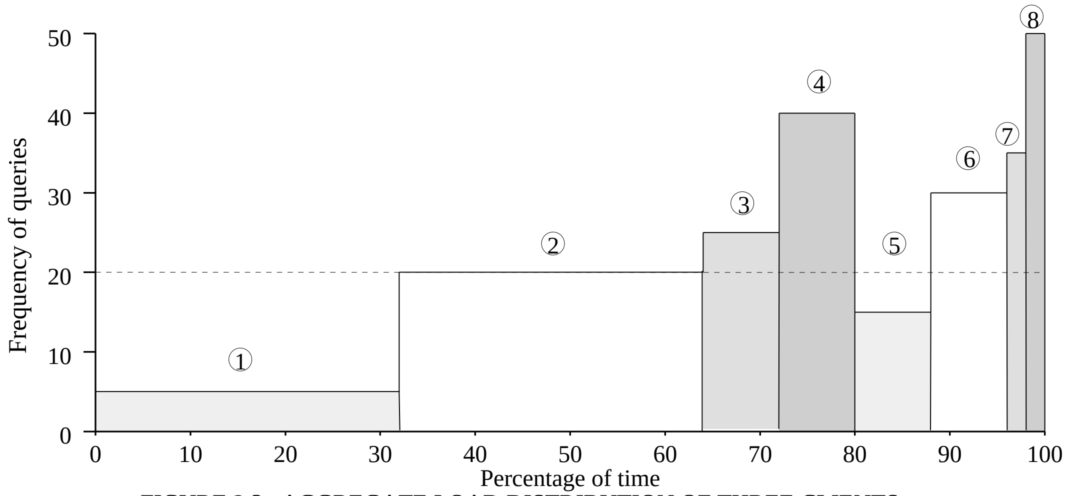

FIGURE 2.3. AGGREGATE LOAD DISTRIBUTION OF THREE CLIENTS

$$\sum_{i=1}^{n} \max_{j=1}^{m}(F_i(j)).$$

The aggregate distribution is calculated by taking all combinations of the *n* clients and *m* modes. For each mode, we determine the probability of operating in that mode by multiplying the probabilities of the combination of modes of the *n* clients. We determine the query rate by summing the query rates of the clients for the given combination of their modes. Because there are *n* clients, each of which has *m* modes of operation, the time to calculate the aggregate distribution using a naive algorithm is $O\,(m^n)$.

We can reduce this to polynomial time by using an approximation. In the approximation, modes are grouped depending on the frequency of requests in the mode, so all modes within a combined group have similar frequencies. A fixed number of modes, *N*, are allowed in the approximation. *N* is assumed to be larger than *m*. The probability of the combined mode is the sum of the probabilities of the calculated modes being grouped together. The frequency of requests in the combined mode is the average of the frequencies of the calculated modes. The use of combined modes does not affect the average load the aggregate places on servers. If the mode containing the peak frequency is kept separate, the combined modes do not affect the peak load calculation either.

Furthermore, we can calculate the aggregate iteratively, first taking two clients, calculating their aggregate load, and adding another and recalculating, until all clients are added. Each step in this process requires $O\,(N^2)$ calculations, using the simple combinatorial method we used above. After the calculations for a step are complete, we recombine the modes of the new aggregate to restrict them to exactly *N* modes. Therefore, if we restrict the number of modes allowed in an aggregate to a constant number, and if we calculate the aggregate iteratively, the time to calculate the approximate probability is $O\,(n \times N^2)$. In a related algorithm[11], it was shown that this approximation can be made arbitrarily close to the real probability of exceeding the server's capacity, by increasing *N*.

2.2. Protection

The purpose of the protection model is to verify that all operations performed by a client on an server are permitted by that server. For example, if a client specification states that it requests routing tables from a server, the specification for that server must state that this client has permission to perform that operation. The protection model includes the property that each request performs a single operation on a single object.

2.2.1. Verifying Protection Conditions

To verify the protection condition, we must show that each client has permission to perform each of its queries. In this discussion, we require the following logical relations:

$Contains_O\,(O_1, O_2)$	Object O_1 contains object O_2.
$Contains_D\,(D_1, D_2)$	Administrative domain D_1 contains system or domain D_2.
$Instan_D\,(D, P, I_{D,P})$	System or domain D instantiates process P with identifier $I_{D,P}$.
$Instan_P\,(I_P, O, I_{P,O})$	Process instance I_P instantiates object O with identifier $I_{P,O}$.
$Req\,(C, I_S, T, I_{S,O})$	An instantiated process or a domain, C, requests to perform operation T on object instance $I_{S,O}$ from server I_S.

Perm $(C, I_S, T, I_{S,O})$	Server I_S permits client instance or domain C to perform operation T, on object instance $I_{S,O}$.

The relation, $Contains_O$, allows permissions to be inherited using the containment hierarchy of the management information. $Contains_D$ gives the relationship between domains and the objects that they contain. Both of these relations are strictly hierarchical − objects and domains cannot contain themselves. The $Instan_D$ relation models the idea that a network management process is a control abstraction that must be instantiated. $Instan_P$ models the way in which a management process instantiates information; multiple instances of information have different identifiers. *Req* describes the permissions necessary in each request that a client makes of a server. *Perm* describes the permissions that a server gives to a client or domain. If permission is granted to a domain, each client in that domain, as defined by the $Contains_D$ relation, is also granted permission. Each of these functions evaluates to true or false for each object or process in the specification.

The functions described above are related by a small set of deduction rules, shown in Figure 2.4.

The *Transitivity* rules formalize the intuitive notion that an enclosing domain or object contains all of the parts of its subdomains or sub-objects. Note that transitivity does not allow deductions from specific cases to generalizations, because containment is strictly hierarchical. The rule of Implied Containment states that when a domain or system, D, instantiates a process, the domain contains that instantiation of the process. The rule of Implied Instantiation states that if a process instantiates an object that contains other objects, the sub-objects are also instantiated by that process. The Implied Object Permission rule states that if a server gives permission to access an object instance, and the object contains sub-objects, then the server also gives permission to access the instantiations of those sub-objects. The Implied Domain Permission rule states that if a server gives a permission to a domain, then anything contained in that domain also has that permission. The rules for defining Unique Identifiers state that instance identifiers are uniquely determined by the process and object in the $Instan_P$ relation, and by the domain and process in the $Instan_D$ relation.

Given these relations and rules, we wish to prove that for every request, there exists a corresponding permission. The problem is simplified by the finite problem space − we only need to prove this condition for a given specification, and not for all possible specifications. This allows an iterative proof method. A simple, centralized method proceeds as follows. First, using the *Perm*, $Instan_P$, and $Contains_O$ relations

Transitivity	$Contains_O(O_1,O_2)$ & $Contains_O(O_2,O_3) \vdash Contains_O(O_1,O_3)$
Transitivity	$Contains_D(D_1,D_2)$ & $Contains_D(D_2,D_3) \vdash Contains_D(D_1,D_3)$
Implied Containment	$Instan_D(D,P,I) \vdash Contains_D(D,I)$
Implied Instantiation	$Instan_P(I_P,O_1,I_{P,O1})$ & $Contains_O(O_1,O_2) \vdash Instan_P(I_P,O_2,I_{P,O2})$
Implied Object Permission	$Perm(C,I_S,T,I_{S,O1})$ & $Instan_P(I_S,O_1,I_{P,O1})$ & $Contains_O(O_1,O_2) \vdash Perm(C,I_S,T,I_{S,O2})$
Implied Domain Permission	$Perm(D_1,I_S,T,I_{S,O})$ & $Contains_D(D_1,D_2) \vdash Perm(D_2,I_S,T,I_{S,O})$
Unique Identifiers	$Instan_P(I_{P1},O_1,I_{P1,O1})$ & $Instan_P(I_{P2},O_2,I_{P2,O2})$ & $I_{P1,O1} \equiv I_{P1,O2} \vdash I_{P1} \equiv I_{P2}$ & $O_1 \equiv O_2$
Unique Identifiers	$Instan_D(D_1,P_1,I_{D1,P1})$ & $Instan_P(D_2,P_2,I_{D2,P2})$ & $I_{D1,P1} \equiv I_{D2,P2} \vdash D_1 \equiv D_2$ & $P_1 \equiv P_2$

FIGURE 2.4. LOGICAL DEDUCTION RULES FOR PROTECTION VERIFICATIONS

obtained from the specification, we employ the Implied Object Permission rule to find the permissions for each object instance. Next, we use the derived *Perm* relations, along with the ones obtained directly from the specification, the $Contains_D$ relation, and the Implied Domain Permission rule to find the permissions for each client process. Finally, we attempt to match each *Req* relation with its corresponding *Perm* relation. If any *Req* relations have no match, the specification is invalid.

There are two problems with this approach. It can be inefficient and violate the requirement that verification be split along domain boundaries. Therefore, we perform the verification in two parts. First, we evaluate the permission relationships within a domain, and then propagate inter-domain requests and permissions to the containing domain.

The verification proof within a domain is performed in the naive way described above. However, before this proof takes place, we remove from the *Req* set the set of requests that are destined for processes in other domains, by checking to see if the server instantiation used in each *Req* relation exists in this domain; if not, we assume it exists in another domain. We call the set that we removed the *external request set*. The remaining *Req* relations are internal requests. We then try to match each remaining *Req* relation with a *Perm* relation, and if these matches succeed, the domain's permission specification is *internally consistent*.

To hide the information specific to a given domain from other domains, we need to generate a new specification and new logical relationships from the set of requests that should be satisfied by permissions from other domains. We also need to make external permissions visible to other domains. If we assume that the internal structure of an administrative domain is not visible outside of that domain, then all requests from any client within a domain will appear to have come from the domain itself. Therefore, the *Req* relations for a domain are derived from the external request set by replacing the client or subdomain identifier in the original *Req* relation with the domain identifier in which it is contained. To avoid the incorrect verification of a specification, we must leave external permissions unchanged, because each server defines its own permissions. If two servers grant different permissions to an object, it is important to differentiate between them when evaluating their clients' requests. This set of external relationships is propagated to the enclosing domain, which repeats the process of internal verification and external propagation. When the top level domain is reached, no propagation takes place, and any unsatisfied requests are determined to be inconsistent.

2.3. Configuration

The configuration model is used to verify parts of NMSL specifications relating to the configuration of network managers. We call each of these a *configuration condition*. This classification includes all parts of a specification not modeled by the capacity or protection models. The configuration conditions have the common characteristic that they are general requirements placed on the entire specification by a single element of the specification. Some examples of configuration conditions are verifying that: the protocol used to communicate between two management processes is the same, the maximum number of routers allowed in a network is less that some maximum, and all requests sent to a process are smaller than the maximum request size accepted by that management process.

Two examples of specifications defining configuration conditions are shown in Figure 2.5. These examples are fragments of a NMSL specification. The first example shows the hardware configuration of a computer on our local network, defining several configuration conditions. The cpu type and operating system version define constraints on the types of programs that can be run on this computer. The interface definition sets conditions for maximum transmission rates and packet sizes.

The second example specifies the requirement that the number of computers that route packets between networks (i.e. those computers that run routing processes), be three or less. This type of configuration condition is important for networking products that have limits on the number of routers allowed on an individual network. They can also be used to enforce administrative requirements. Limiting the number routers on a network to one, for example, would enforce the requirement that that network cannot be split into two physical subnetwork without intervention from the central administrator.

These conditions can also be used at runtime to configure network management processes. For example, by listing the interfaces a computer has and the networks to which it is attached, a network management server can be configured to monitor these interfaces. Different instantiations of the management process can be configured in different ways, depending on these conditions. This use is part of the

```
system "dip" ::=
    cpu mips;
    opsys ultrix version 2.1;
    interface se0 {
        net wisc-twisted;
        type ethernet-csmacd;
        speed 10 Mbps;
    }
end system "dip".

domain "wisc-cs" ::=
    requires {
        count(process Router) >= 1 and
        count(process Router) <= 3;
    }
end domain "wisc-cs".
```

FIGURE 2.5. EXAMPLES OF CONFIGURATION CONDITIONS

prescriptive role of NMSL, mentioned in Section 1.

To verify the configuration conditions, we must show that each of the conditions in a specification are met. The method used is similar to that use for verifying permission conditions. To verify conditions concerning amounts, or those concerning numbers of entities (*e.g.* objects or systems), we employ equations, in the manner similar to that used for verifying the capacity of the system.

2.4. Domains

Our model thus far has implicitly assumed that we will evaluate the specifications in a single, central location. This centralized method could cause two problems. First, the number of logical relations to evaluate would be quite large. Second, it would require that each organization provide detailed information about their internal computing environment. Both of these problems occur at the domain boundary, and share a common solution, information hiding. The goal of information hiding is to summarize the specification at domain boundaries and propagate a small subset across the boundary.

The first problem, that of large numbers of logical relations, occurs when specifications need to be propagated across domain boundaries. This is because logical relations can reference other domains, for example, a relation describing a client in one domain querying a server in another domain. In Section 2.1, we described a method for summarizing the load clients place on a server in another domain. This method involved calculating the aggregate load for the clients, and also an approximation algorithm that reduces the number of modes and the time needed to calculate the aggregate. As we will show, summarization can be applied to other parts of a specification as well. The summarization will result in a subset of a domain's specification being propagated across the domain boundary. The summary of one domain's specification is propagated to the domains that contain that domain. The summarization step can be repeated on the specifications of each of the containing domains. In this way, at each step of the distributed verification process, only the summary of external references needs to be propagated across domain boundaries. This results in much less copying of information and better performance than a centralized verification method.

The second problem, requiring an organization to provide the structure of its internal computing environment, is a privacy issue. Privacy is important for several reasons. A network administrator may want to change private portions of a specification without notifying a central authority. Providing the internal structure may violate the security constraints of the organization. Furthermore, some details of the internal structure may be proprietary.

One solution for these problems is to provide an information hiding mechanism. This mechanism determines which parts of a domain's full specification are reflected in its external specification, and which parts remain private. The private parts of a specification can be verified locally, with a domain. The external parts must be propagated to other domains. Furthermore, if domains are organized in a hierarchical manner, the most common organization, we can restrict the propagation of external specifications to parent domains, reducing the amount of copying necessary.

Information hiding is necessary to preserve the autonomy of each domain, and ensure that private management information is not visible to the outside. It can also prevent information explosion. We employ three methods for providing information hiding. First, we include only those parts of a domain's specification that reference another domain. Second, we remove private and common information. Third, we summarize the remaining information.

When determining which parts of a domain's specification to include in the external specification, we consider the low level, logical relations, not the high level NMSL specification. Those relations that reference only objects that are in the local domain and cannot be referenced by another domain, can be excluded from the external specification. Examples of such relations are those that concern the frequency of requests a client in the domain makes to a server within the same domain, the permissions involved with those requests, and the configuration conditions of individual systems within the domain. Relations concerning process specifications can also be excluded; all that is important to the external specification is the behavior of the instances. We must include relations that have an indirect effect on other domains. An example of an indirect effect is if a client queries a server that serves both clients both inside and outside the domain.

We can exclude those relations that are private, and cannot be referenced by another domain, and those that are common, relations that are already known by other domains. Removing private and common information requires knowledge of where a specification originated. This knowledge is most important for removing object specifications from the external specification (*e.g.* the standard Management Information Base (MIB)). Private information must originate within the domain. Common information is information that all domains include in their specifications. If we assume that the object specifications are derived from a central source, specifications concerning objects may be excluded from the external specification of a domain, because the parent domain's specification will also include all of the objects. Private object specifications can also be excluded, because no other domain will reference them. Some information of a private nature, such as the interaction of clients and server within a domain, must be included in the external specification, if they have an indirect effect on the external specification, as we showed above.

To summarize the capacity specifications of a domain, we determined the aggregate load that the domain's client processes place on other domains. To do this, we must determined which clients contribute to the domain's aggregate load, and then calculate the aggregate load. Such clients include those that refer to servers in another domain. Clients that refer to a server that serves both the local the other domains must also be included in the summary; these clients cannot be considered private. At this point, the only remaining logical relations for clients are those that make external references. Given these external references, we use the aggregation method described in Section 2.1 to calculate the aggregate load.

We cannot hide the specifications of individual servers in the way we described above, because any aggregate of the servers' capacities would lose information about the individual servers' capacities. This would allow invalid specifications to pass through the verification phase, despite inconsistencies. For example, if all of the clients query the same server in a group of servers, they could overload that server, but if the servers' capacities are aggregated, this inconsistency will not be recognized.

To summarize the protection specifications, we propagate relations listing the objects referenced, but specify the domain, not a process instance, as the initiator of the reference. Once again, the permission relations must be propagated unchanged. Summarization of configuration specifications follow the rules we described for the capacity and protection specifications.

3. APPLYING THE CONSISTENCY MODEL

In practice, network management systems perform high level operations, such as detecting error and faults, and noting exceptions. These operations often cause messages to be sent from a network management server to an application, such as a network operation center (NOC) tool, so that the network operator can take appropriate action. While this is a small part of what a network management system can do, it is the most common current application. This section shows an example of one way error reporting can be performed, the way to specify the example in NMSL, and the relationship between the specification and the three verification categories.

The NMSL system supports the specification of error reporting events and the way in which they are reported. Error reporting is not given a separate category in our consistency model. Instead, it has components of all three of the categories we discussed.

To see how errors propagation is handled in our Consistency Model, we use a simple example. Consider a router connecting two networks, and a management station (another computer) on one of these networks. The router, in addition to routing packets, runs a network management server or agent. The network management stations runs several network management applications – these all communicate with the agent via SNMP[4]. The agent supports trap management, as defined in the SNMP standard, and sends trap notices to a trap management application on the management station. A specification of this example is shown in Figure 3.1. The gateway is the system `gw` and the management station is called `dip`. The agent and application processes are `snmpd` and `snmp_trap_handler` respectively. In this specification, we have set the rate of interaction to be at most one trap message per hour (1 qph).

For traps to be sent correctly, several conditions must be met:

(1) The agent must specify the recipient of the trap messages (the application running on the network management station).

(2) The agent must give the recipient permission to view any trap related data objects.

(3) The recipient must be interested in receiving trap messages, *i.e.* it must permit the agent to send it the traps.

(4) The agent and the recipient must speak the same protocol and agree on a rendezvous point (e.g. an IP port number).

(5) Packet sizes must be within acceptable bounds.

(6) The rate of interaction (sending of trap messages) must be within the limits of the configured system.

These requirements were arrived at by examining the contents of a `TRAP-PDU` message in SNMP, which is sent from an agent to an application, as well as general requirements of network management interaction.

Some of these requirements are difficult to determine, especially (6), but a reasonable value can be determined for interaction rates based on examining the mean time between failure characteristics of the the routing hardware. Obtaining such information for the agent itself is not a subject of this paper.

A brief inspection of the requirements given above show that they fall into the three categories, capacity, protection and configuration. Conditions (1), (2) and (3) are all specified using the protection constraints described in Section 2.2. Conditions (4) and (5) are specified using configuration constraints. Condition (6) is a capacity constraint for handling these traps. These six requirements are all present in the specification in Figure 3.1. Condition (1), the recipient of trap messages, is shown in the specification for the system `gw` and specification for the `snmpd` process it runs. In this case, it specifies the recipient, `dip` as an parameter to the `snmpd` process specification. Condition (2), giving permission to examine trap related data, is specified in the `sends` clause. Here, no data is sent, so no additional permission is allowed. (3), where the recipient allows traps to be received, is specified in the `receives` clause. Conditions (4) and (5) define the packet sizes and the protocol and appear the `sends` and `receives`. Condition (6), the capacity, is given in the `rate max` subclause of the `sends` and `receives` clauses.

As this example shows, the NMSL specification written by a network administrator need not be divided into these three parts. However, for the purpose of verification, the compiler/verifier takes the input, high level specification, and generates constraints in these three forms.

4. Implementation and Performance

We have implemented a verifier for the Consistency Model described in the preceding section. Verification is a two step process. First, a high level, NMSL specification is compiled into a low level set of logical relations, like the ones described in Section 2. Next, these logical relations are passed to a proof checker that uses the rules we described to find inconsistencies in the specification. After a short description of the the compiler and verifier, we present the performance of the current implementation.

The compiler is written in C, and provides an interpreted extension language. The compiler's internal parser enforces only the syntactic structure of the specifications. An early version of the basic structure

```
system "gw" ::=
    cpu cisco; opsys cisco version 2;
    interface ie0 {
        net wisc-research;
        type ethernet-csmacd; speed 10 Mbps;
    }
    interface ie1 {
        net wisc-twisted;
        type ethernet-csmacd; speed 10 Mbps;
    }
    supports mgmt.mib;
    process snmpd("dip");
end system "gw".

system "dip" ::=
    cpu mips; opsys ultrix version 2.1;
    interface se0 {
        net wisc-twisted;
        type ethernet-csmacd; speed 10 Mbps;
    }
    process snmpd_trap_handler("gw");
end system "dip".

process snmpd(HOST: string) :=
    supports mgmt.mib; -- entire MIB subtree

    exports mgmt.mib {
        to "wisc-cs";
        access ReadOnly;
        rate max { mode 1 1.00 10 qps; }
    }
    sends traps {
        to HOST; using protocol "snmp"; port "snmp-trap";
        data none;
        rate max { mode 1 1.00 1 qph; }
      provides { "packetsize" <= 484 octets; };
    }
end process snmpd.

process snmp_trap_handler(HOST: string) :=
    receives traps {
        from HOST; port "snmp-trap"; using protocol "snmp";
        data none;
        rate max { mode 1 1.00 1 qph; }
      requires { "packetsize" <= 1024 octets; };
    }
end process snmp_trap_handler.
```

FIGURE 3.1. EXAMPLE OF AN ERROR PROPAGATION SPECIFICATION

of NMSL specifications is described in [5]. This syntactic structure is a list of clauses, with the ability to group clauses into blocks. Examples of NMSL specifications are shown in Figures 2.5 and 3.1. The semantic processing is performed by action routines written in the extension language The action routines also perform the code generation and other tasks, such as symbol table management. Generally, each clause in the specification corresponds to a capacity, protection, or configuration condition. Given a clause, an action routine first performs the semantic checks on the clause, and then generates the appropriate logical relations corresponding to that clause. The use of our extension language has allowed more rapid implementation of the specification language, and reduces the turn-around time when debugging the semantic routines.

In our implementation we represent logical relations as statements of CLP(**R**)[1]. CLP(**R**) is a Constraint Logic Programming language that provides a logic programming model similar to PROLOG, but with a more general proof mechanism than that used by PROLOG. The CLP(**R**) mechanism includes the ability to solve equations over the real numbers, which is useful in proving the capacity conditions. Given these logical relations, we use a set of deduction rules, also written in CLP(**R**), to determine if the capacity, protection, or configuration conditions are violated in the specification. Violations are reported to the administrator.

Several issues are important to the performance of the NMSL verifier. We need to quantify the sizes of the input specifications that the NMSL verifier is expected to process. Because we are concerned about information explosion at the domain boundaries, we need to determine how much information can be hidden within a domain, and how much must be propagated. The time needed to verify a specification is also of importance. To determine this, we must look at the execution time of the NMSL compiler and the time it takes CLP(**R**) to verify a specification.

At the time this paper was being written, the automated network management system in our department was still in a early state. Because of this, we have written specifications for the for a representative management structure based on part of our current network configuration. The specifications were written for an SNMP environment.

We use these specifications to study various aspects of the performance of the NMSL system. The specifications demonstrate how the complexity of a network management system affects the size of a NMSL specification. We also use the specifications to determine the effectiveness of summarization on reducing the size of external specifications. The effect of summarization is important, because the smaller the external specification, the less effect it will have on total verification time. By total time, we mean not only the time it takes to verify the internal specification, but also the time it takes to verify those domains to which the external specification is propagated. The larger the external specification, the more time it will take to verify it. We measure the total time it takes to verify our test specifications. This gives us an idea of the time it takes to verify NMSL specifications in general, and the ability of NMSL to specify and verify large network management systems.

The tests are straightforward. For each specification, we determined its size based on a simple line count. Each line corresponds, on average, to a clause in NMSL. We then compiled the specification and counted the number of logical relations it took to represent the NMSL specification. We also inspected the logical relations and determined the size of the internal and external specifications, in terms of their size in logical relations. Note that our implementation does not currently perform this separation automatically. Last, we used CLP(**R**) and the deduction rules we described in Section 2 to verify the specifications, and measured the time needed for verification.

The results of our tests are shown in Table 4.1. The first test is an empty file. This test shows the startup costs for the compiler to process the extension language statements. The second test is a specification of processes and systems within a domain, but describes no data objects. This test file shows the time it takes to compile and verify a specification with no data objects. The third is a full specification, including processes, systems, data objects, and domain groupings. By comparing the results of the second and third tests, we can determine the effect the number of data objects in a specification has on its verification time. The second and third tests included 7 system specifications, 3 process specifications, 2 server and one client, and 2 domains. The server processes are instantiated on each of the systems, the client at one central location. Of the domains, the first includes 6 of the systems, the second, the remaining system. The object specifications in the third test are those defined in the complete RFC1066 MIB[10]. The internal and external parts shown in Table 4.1 are divided using the criteria discussed in Section 2.4. These tests were run on a Sun 4/110, running SunOS 4.0.3 with 8 Megabytes of main memory. The file system buffer cache was primed before the tests were run, so disk activity was not a factor. The verification time for Test 3 is shown in minutes and seconds.

	Test Number 1	2	3
NMSL Specification size (lines)	0	210	1215
Number of Logical Relations Total	0	191	569
Internal Part	0	174	1198
External Part	0	17	17
Compile Time (sec)	1.22	2.06	3.52
Verification Time (sec)	0	0.98	3:50.30

TABLE 4.1. Performance of the NMSL Verifier

Several conclusions can be drawn from these simple tests. The sizes of specifications are intended to be large; the specification for a single domain will be well over 1000 lines long. The number of logical relations for a domain correspond roughly to the size of the input specification (Test 3 includes empty and comment lines). More importantly, the size of the external specification is kept small compared to the internal specification's size. Because the data object specifications in Test 3 came from a standard MIB, we were able to exclude them from the summary as well. Therefore, our summarization methods seem to be effective.

The compiler has a reasonable but noticeable startup overhead, due in part to the use of the interpreted extension language. However, after this time is factored out, it processes specifications quickly. A very important result is the effect the number of data objects in the specification has on the total verification time. In Test 2, the verification step executed quickly, while in Test 3, many data objects needed to be checked for the protection conditions. Because there are so many data objects in the SNMP MIB, this becomes the main factor in the verification time. The implementation in CLP(**R**) does not prove the equivalent of a lemma, and reuse such results throughout the rest of the proof. This results in re-proving partial results each time they are needed. Some enhancements were made to the implementation to reduce this effect, but additional work is needed in this area.

5. Summary

In this paper, we discussed the problems caused by the complexity and autonomy in modern network management systems. These problems are the result of larger and more complex networks. To solve the network management problem for large networks, the management system is forced to into a distributed rather than a centralized model. The distributed nature of managing large networks is exacerbated by administrative domains. Administrative domains reduces the coordination of configuring the network management system, and increases the chance that parts of the system will not be configured correctly.

To solve these problems, we use a formal specification language. The specification language, NMSL, addresses these problems by providing a way to formally specify the configuration of a network management system, and a mechanism to automatically verify a specification. We use a model for network management systems, based on four concepts: administrative domains, systems, management processes, and managed objects. We described the way a network management system fits into this model, and the method that NMSL uses this model to verify specifications. This method divides the verification problem into capacity, protection, and configuration conditions. Separate methods are used to verify each of these conditions.

We also distribute the verification problem along administrative domain boundaries. We use summarization to reduce the size of external specifications that are propagated across these boundaries. The distribution and summarization provided by the NMSL verification model are very important to preserving the autonomy of individual domain, and reducing the overhead of copying specifications across the network.

The current results of our work is encouraging. Our models for capacity and protection verification are well defined. We use closed-form equations are used for determining the capacity of servers. We also use a simple logic to verify protection conditions. Similar methods are used for proving the capacity conditions.

We have used the NMSL verifier to verify some representative specifications. Our tests show that specifications for small domains can be processed by the NMSL verifier in reasonable time. Our initial performance tests make several important points. The mechanisms we proposed for summarization cause a great reduction in the size of external specifications. The verification step also executes in a reasonable time for simple specifications. However, the performance of the verifier is affected greatly by the number of data objects present in the specification. This was caused by re-proving partial results in the verification proof.

Several issues remain to be addressed. The performance of the verifier for large numbers of data specifications is not as good as we had hoped. The cause of the problem has already been determined, but we need to investigate ways to avoid the problem. The current implementation of the verifier does not include a way to divide a specification into its internal and external parts. We need to implement this in the way described in Section 2. Next, we are turning to the other role of the NMSL system, which uses a verified specification to configure the network management system. Work is already underway to set up an environment for testing the configuration role of NMSL.

NMSL and the techniques described are also appropriate for specifying and managing distributed systems other than network management. We plan to investiate applying NMSL to these other systems and services.

REFERENCES

[1] N. Heintze, et al, *The CLP(R) Programmer's Manual*, Dept. of Computer Science, Monash University, Clayton, Victoria, Australia (1987).

[2] K. P. Birman, ''Replication and Fault-Tolerance in the ISIS System,'' *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pp. 79-86 Orcas Island, Washington, (December 1985).

[3] L. Bronner, ''Overview of the Capacity Planning Process for Production Data Processing,'' *IBM Systems Journal* **19**(1) pp. 4-27 (1980).

[4] J. Case, M. Fedor, M. Schoffstall, and J. Davin, ''A Simple Network Management Protocol,'' RFC 1157, IETF Network Working Group (May 1990).

[5] D. L. Cohrs and B. P. Miller, ''Specification and Verification of Network Managers for Large Internets,'' *ACM SIGCOMM 89*, pp. 33-44, Austin, TX, (September 1989).

[6] Information Processing Systems – Open Systems Interconnection, ''Specification of Abstract Syntax Notation One (ASN.1),'' ISO 8824, International Organization for Standardization (December 1987).

[7] Information Processing Systems – Open Systems Interconnection, ''Management Information Service Definition,'' ISO DIS 9595/2, International Organization for Standardization (1988).

[8] Information Processing Systems – Open Systems Interconnection, ''Management Information Protocol Definition,'' ISO DIS 9596/2, International Organization for Standardization (1988).

[9] Information Processing Systems – Open Systems Interconnection, ''Basic Reference Model Part 4 – OSI Management Framework,'' ISO DIS 7498/4, International Organization for Standardization (1989).

[10] K. McCloghrie and M. Rose, ''Management Information Base for Network Management of TCP/IP-based Internets,'' RFC 1156, IETF Network Working Group (May 1990).

[11] D. C. Verma, Private correspondence.