A SPECIFICATION LANGUAGE FOR MULTI-DOMAIN NETWORK AND DISTRIBUTED SYSTEMS MANAGEMENT

by

David Lawrence Cohrs

Computer Sciences Technical Report #1044

September 1991

A Specification Language

for

Multi-Domain Network and Distributed Systems Management

by

David Lawrence Cohrs

A thesis submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN — MADISON

1991

ABSTRACT

The size of computer internets is increasing the need for automated network management systems. These network management systems are divided into administrative domains, with independent administrators managing each domain and the exchange of management information between domains. The large number of interacting domains needs some way of coordinating their activities. This research addresses the complexity of managing such a system by providing a language to formally specify the system's configuration, and by providing a tool to incrementally check the specifications for consistency and install consistent specifications in a running management system.

The language, called NMSL, includes facilities for describing the interactions between components of network and distributed management systems. Interactions are described in terms of the protection, capacity, and configuration constraints placed by an administrator on communication within and between administrative domains.

A collection of NMSL specifications can be checked for consistency. We describe both a *centralized* and an *incremental* method for evaluating consistency. We present a detailed description of logic rules for evaluating consistency of NMSL specifications centrally. We then show how a centralized proof can be partitioned and performed incrementally. An important result of performing an incremental proof is that we can divide the proof along domain boundaries and only propagate small summaries of per-domain information between domains. The incremental method is shown to be better than the centralized method for reasons of performance and preservation of autonomy and privacy of administrative domains.

These methods are prototyped in an implementation. With this implementation, we show that NMSL is powerful enough to specify existing network management systems. We show that our approach to incremental consistency checking meets the goals of improving performance and preserving autonomy. We also show how a consistent specification can be installed in a running management system. We discuss the implications of automatic installation and describe problems encountered when trying to enforce specifications. The implementation demonstrates the feasibility of our ideas, and suggests areas for future research.

ACKNOWLEDGMENTS

Bart Miller, my advisor, has been an unlimited fountain of knowledge during my tenure as a graduate student. His patience and, just as often, his impatience has helped shape my research skills and has motivated me beyond anything I thought possible. I owe him my deepest gratitude.

My thesis committee, Profs. Marvin Solomon, Lawrence Landweber, Raghu Ramakrishnan and Michael Byrd, also deserve my thanks. Special thanks go to Profs. Solomon and Landweber for taking the time to carefully read this dissertation and improving the presentation. Their comments and suggestions during this research also proved invaluable. Thanks also to Profs. Miron Livny and Ken Kunen for their help in clarifying my evaluation methods.

Above all, Laura Berg deserves my thanks. She patiently listened as I babbled on about my research. Her love and support kept me going when I was ready to give up. She even tried to be understanding when I gave the computer more time than I gave her. Laura, I love you, and I hope that I have made you proud.

TABLE OF CONTENTS

TABLE OF FIGURES

Chapter 1

Introduction

Computer internets are growing in size and complexity, and the need for automated network management systems is growing as well. If the network is small, the task of network management can be performed by a human administrator, using simple, *ad hoc* tools. In a network with two or more independent administrators, managing the network management system becomes a significant problem in itself. This thesis addresses the complexity of managing the network management system for multi-administrative domain internets through the use of a language to specify the configuration of network management systems. The structure of this language is suitable for the general problem of controlling distributed systems management as well, as an extension to network management. The methods described in this thesis allow autonomous administrators to formally specify the intended configuration of their network management systems, formally check these specifications for errors, and then check these specifications for consistency with the specifications written by other administrators. Furthermore, these methods take advantage of knowledge about the specifications, both in their structure and in the possible relationships between specifications written by different administrators, to improve the efficiency of the error checking. These methods are prototyped in the Network Management Specification Language (NMSL).

The job of a *network management system* is to gather data about a network, the attached devices, and the programs running in the network. It must present this data to network operators or administrators, and allow them to *monitor* and *control* the network. Monitoring involves watching the network's behavior, and controlling requires the ability to modify the behavior of parts of the network, including process behavior.

A network management system is similar to other networking or distributed applications, in both design and difficulty of maintenance. The system instantiates processes throughout the network. The processes communicate via network management protocols, which must be configured correctly. For example, 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.

Additionally, an internet, and therefore, the network management system, is generally divided along the lines of *administrative domains*[9]. An administrative domain is a group of networks and attached equipment that is controlled by a single organization, such as a company or a university. Administrative domains can be hierarchical. For example, a company can be divided into divisions, and the divisions into departments. Each department is responsible for administering its own networks, but the interconnection of the departments is under the control of the division, or perhaps the company's computing services department. Similar hierarchies also exist in academic settings. These corporate and academic networks can be connected together in an internet. Existing internets typically have a backbone, possibly hierarchically organized, with a backbone per region and a backbone connecting the regions, to transmit data between organizations. This backbone may be administered by a separate organization. Organizations can also coordinate their own connections to other networks without using a backbone network. Hierarchical administrative domains exist today in such networks as TCP/IP Internet.

Multiple administrative domains introduce several problems to managing an internet. First, they cause an inherent distribution of control. Each administrator wants to control the networks and devices in their domain, and in general will not want to give up this control to a central network management organization.

Second, because the control must be distributed, there must be some way in which the multiple domains can cooperate to perform global management functions, such as routing, name space management, configuration management and accounting. Cooperation is difficult because separate administrative domains are often hostile to one another; the network manager in one domain might not trust the statistics and control requests coming from a network manager in another domain.

Finally, there is a lack of responsibility, especially in datagram networks. When a user in one domain sends datagrams into another domain, it is difficult to trace the origin of these packets, introducing security and control problems. The users in one domain of an internet are not held responsible for the problems they cause in other domains.

The goal of this research to provide methods for network administrators to effectively manage their network management systems, both within a single administrative domain, and to coordinate the management of multiple administrative domains. The thesis of this dissertation is that a formal specification language combined with a system built around this specification language can be used to achieve this goal. The important contributions derived from this work are:

- A model for network management systems, and a definition for the correct operation of a network management system.
- A *descriptive* specification language allowing network administrators to describe both the *internal* operation of a single administrative domain's network management system, and the interfaces between the various network management systems.
- A *prescriptive* meaning for the language. The specification can be translated into control commands that configure the various network management systems. These commands set individual configuration parameters necessary to achieve the global management goals. These control commands can be based on the network management standards such as the OSI organizational model[23], or the TCP/IP management framework[5].
- The language is high level, and relatively easy for non-experts to understand. This allows non-experts to write specifications.
- We provide the administrator with a way to find errors in their network management system before such errors cause problems under normal operation. We locate inconsistencies in specifications, or indications of incorrect behavior, by mapping the specification onto a logic programming language, and verifying its

consistency.

- We have extended the verification method to include administrative domains, and devised an incremental proof method that preserves the domains' autonomy and provides for information sharing and hiding. In addition, this method can reduce the time it takes to verify the multi-domain specification.

We are not directly addressing the important issues of security or authentication. We assume the existence of a suitably strong authentication mechanism that can be used by the NMSL system. We also do not attempt to define the format of the network management data or the low level management protocols themselves. We assume the existence of standard management data representations and protocols. In the NMSL prototype, we depend on the TCP/IP Management Information Base (MIB)[27,28] and use the Simple Network Management Protocol (SNMP)[5].

This dissertation is organized as follows. Chapter 2 describes related work in network management systems and standards, as well as work from the specification language literature. In Chapter 3, we describe our model for network management systems. We also present an overview of our specification language, and the system built around the language.

Chapter 4 discusses the language, its grammar, and its semantics. The facilities provided by the language for specifying network management systems are described and shown in examples.

Chapter 5 describes a method for mapping NMSL onto a logic programming language. Given this mapping, we describe a centralized proof evaluator. We then extend this evaluator to operate incrementally. The various properties of this mapping and the evaluation methods are detailed.

Chapter 6 examines the prescriptive aspects of the language, i.e., how a specification can be changed into a format that can be used to control the network.

Chapter 7 describes our prototype implementation of the NMSL system, including the performance of that prototype, and the studies we have performed to show the utility of this system. Chapter 8 concludes with a summary of this research and its contributions, and offers suggestions for future research in this area.

The Language Reference Manual for the NMSL prototype is in Appendix A.

Chapter 2

Related Work

This research is based on work in the areas of both network management and formal specification languages. Network management has been important in the telecommunication industry for decades, while in the computer industry, networks have only recently become complex enough that *ad hoc* methods are no longer sufficient. Specification languages have been used in many areas of computer development, for specifying both hardware and software designs. Some specification languages have been designed specifically for network management. These languages will be addressed in particular, and we will show that while they address important design issues, they are not well suited for specifying configuration issues.

2.1. Network Management

The problem of network management is to provide a network administrator with the equivalent of an operator's console for monitoring and controlling the operation of their network. Current research into providing this network management function can be divided into three areas: management architectures and protocols, expert systems, and interfaces. Management architectures and protocols include defining the software components of network management systems, designing data models for network management, defining protocols for exchanging management data in the system, providing security to the protocols, and developing languages for specifying management protocols and systems. Research into expert systems for network management is an attempt to find ways for the system to diagnose network problems automatically, reducing the workload for the network administrator. Interface developers concentrate on providing the actual console for the network administrator, often using workstation technology. Our work in network management addresses the first area, defining both a model for multi-administrative domain management and a specification language for such systems. As such, we will concentrate on work in defining management architectures and protocols, and their support for multiple administrative domains.

Current research efforts are investigating the management of today's computer networks, and standards are available for the low level monitoring and control functions in the network. Some of these research projects also partially address the issue of administrative domains. We will discuss the administrative aspects of these research projects, especially multiple administrative domains and system configuration. Papers describing early network management systems[7,29,36] discuss problems relating to administrative domains, but do not offer any solutions. These systems are precursors of the current management systems, and will not be mentioned further. Among the current systems that deal with administrative domains are network management systems for the TCP/IP Internet[5], the OSI networks[23], AT&T's networking products[22], and IBM's SNA networking product[30]. The Internet and OSI network management systems are meant for use in large, distributed internets, with no central administration. The AT&T and IBM network management products primarily address the needs of a single customer organization, and do not include general support on the problems of administrative domains. Both the AT&T and IBM products include a mechanism for delegating control, and for allowing different administrators different amounts of control over the network. The administrative aspects of these systems are detailed below.

The Simple Network Management Protocol (SNMP)[5], is a draft standard for managing TCP/IP based internets and is being studied by members of the Internet Engineering Task Force (IETF). Implementations are currently being built and tested in the TCP/IP Internet. This protocol defines the low level interactions between the managed ***network elements*** and entities that perform the actual management functions. The managed network elements run a program called the management ***agent***, and the management functions are performed by ***applications***. As the name states, the protocol is simple, defining only a small number of messages. These operations allow applications to retrieve and change objects (called variables) in a network management database in an agent. The agent processes are located on the remote network elements. SNMP employs polling techniques as the primary method for receiving data from remote network elements. Polling is important to the scalability of the protocol, because when polling is used, no management data will be sent unless it is requested. SNMP also provides exceptional event messages, called *traps*, for notifying applications of exceptional conditions, such as the restart of an agent. Traps can also include management data, but their use is intended to be infrequent.

SNMP allows for distributed control of an internet and assumes that there will be many independently operating management entities throughout the network. These management entities are divided into *communities*. Every variable in the database is assigned an access mode. Each community has a view of this database, and an SNMP access mode. The access mode may be either reading or writing, or both. A management entity in a given community can access a specific variable if the variable is in that community's view and both the variable's access mode and the community's SNMP access mode allows the access. This protection mechanism allows flexibility in deciding how a remote domain of administration, such as a community, can access data from a network element's database.

The OSI network management architecture[23], and the International Standards CMIS and CMIP[17, 18] also include support for administrative domains in its current definition. The OSI organizational model assumes that management of the network will be distributed across different domains of administration. Each management domain communicates with other domains via *ports*. These ports may be symmetrical or asymmetrical in the data that may be transferred between the domains. Management domains may also be nested, and the internal features of domains may be hidden from the outside applications. Configuring network managers based on the OSI model could be complex and error-prone, due to the model's generality. Providing a formal specification of the correct configuration could reduce these errors.

StarKeeper[22] is a network management system for managing AT&T's Datakit virtual circuit networks. Its implementation is basically an extension to the existing administrative support available on each Datakit node;

as such, its protocol for accessing remote nodes is more complicated than SNMP. StarKeeper is a centralized management system, assuming that most management activity for the entire network will originate from a single management location. It includes facilities to allow the central network administrator to delegate some control functions, such as the configuration of simple network devices (like multiplexors) to the support staff in subordinate locations. While delegation acknowledges that administrative functions are not always centralized, this mechanism is less general than the SNMP access controls, and may not easily extend to a large, diverse internet.

NetView[30], the IBM SNA network management product, is another fully developed management system. NetView is a centralized management service for SNA networks, but also includes methods for integrating non-SNA components into the system. It provides a central operator with the ability to fully monitor and control a network and all of the devices attached to it. NetView also allows different operators to have different privileges, allowing, on an operator by operator basis, restrictions on which parts of the network the operator may monitor, and the types of control and change operations they may perform. There is also a mechanism for handling management of networks that cross administrative domain boundaries, but the details of this support are not given in the literature.

Other research on network management specifically addresses administrative and security issues, rather than overall system design. Once again, these issues have been addressed on a small scale and at the lowest level. In both CERNNET[11] and Wilbur's local area management model[37] administrative and organizational issues are addressed, but only within a small, local area environment. Both these papers concentrate on the mechanics and implementation of such a system. Interorganizational administrative requirements are mentioned, but not addressed by either paper.

These systems and research projects illustrate the current directions in network management, especially in their support for multiple administrative domains. In both StarKeeper and NetView, multiple administrative domains are possible. However, the management is a centralized, extended to allow data sharing via delegation, where a central domain delegates control over part of their network to a subordinate domain. This structure does not match the way the present NSFNET[6] operates, for example, where control is distributed by default, and administrators that share data maintain control over their own data. The OSI architecture, while proposing a rich set of operations for the sharing of management data, does not include a proposal or policy for implementing these operations. The IETF mechanism is simple and well defined, but does not attempt to define high-level abstractions, such as those described in the OSI architecture. As such, the IETF mechanism provides a simple protocol and a way to specify to which domain multiple clients belong (via the community identifier). In both of these standards, methods for determining the policy for the administrative domains, such as what data can be shared and by whom, are left up to individual implementations. None of these systems include a protocol for coordinating the configuration of the various network managers under the control of different administrators. Also not defined are high-level specifications of the required performance of the network and the network managers. These are the areas that NMSL specifically addresses.

2.2. Formal Specification Languages

Formal specification languages are used in many environments, most notably for specifying programs and protocols. These languages come in two basic types, *algorithmic* specifications and *axiomatic* specifications. Algorithmic specifications are similar to traditional programming languages, in that they specify the algorithms used to implement the system being specified. However, there is formal meaning associated with each statement in an algorithmic specification. Because of this formal meaning, we can easily prove properties about the algorithmic specification, such as its correctness. Axiomatic specifications list properties of the system being specified rather than the algorithms used to implement them. Therefore, many implementations are possible for a single specification. Some specification languages are *executable*. An executable specification is one that can be executed to perform the task described by the specification. Both algorithmic and axiomatic specifications can be executable. Our interest in specification languages, however, is in their use as part of a larger tool, rather than in the correct way in which a specification language should be designed.

A few of the better known program specification languages are Ina Jo[2], PAISLey[39], and Larch[38]. We discuss the features of these languages that are important to designing our network management specification language. We also mention three other recent program specification languages, Gist[10], PLEASE[34], and Anna[26], that share features in common with our work. The protocol specification languages LOTOS[16], Estelle[15], and SDL[12] have been designed for specifying network management issues. Their capabilities are similar to each other and, as we will show, address issues different from those addressed by this research.

Ina Jo and PAISLey support specification of the interactions between multi-process programs. Both languages fall into the algorithmic category. The ability to specify timing properties is important in a network management system, where overloaded resources can be a problem. A network management system is inherently distributed among the various hosts, gateways, and network management stations, so the interactions between these components must be specified. The ability to handle requests in real-time, and the bandwidth required to transmit requests and responses are also important properties in a network management specification. Both these languages could be applied to the timing and bandwidth requirements of network management, but they do not address the problem of administrative domains.

Gist is similar to Ina Jo and PAISLey in that it is used to specify the interactions and operation in a system of interacting components. It differs in that it is an axiomatic language. This approach is appropriate for specifying multi-domain network management systems, because axiomatic languages allow us to concentrate on the interaction between the components of the system without knowing their internal operation. This property can be used to specify management components where only the external behavior is known (e.g., a binary program purchased from a vendor) or to hide information between administrative domains. Gist is not directly applicable to our network management problem because the language is lower level than we wished to present, and because it did not directly support administrative domains.

Larch also takes the axiomatic approach. In addition, it provides a two-level model for specifying programs. The lower level describes the underlying data abstractions, and the upper (interface) level describes how state changes occur. Larch also allows, in its interface specification language, the specification of interactions between independent processes. The two-level model gives Larch generality. Separating the specification of abstractions from state or interaction specifications can also be used for specifying network management. Although this concept is one we use, Larch is a programming specification language, and is not tailored for network management specification.

PLEASE and Anna include mechanisms to prototype or test the programs with which specifications are associated. As such, they fit into the category of executable specifications. PLEASE allows the programmer to convert specification into a prototype. The user can then verify the specification by testing its prototype against a battery of test data. Anna is designed specifically for use with Ada and allows the specification to be translated into run-time checks in the Ada language. These run-time checks are used to verify that the specification is being met. Once again, these languages are meant for programming language specification, and do not include mechanisms for describing administrative domains.

Estelle, SDL, and LOTOS are specification languages used by the OSI community for specifying network protocols and applications. SDL is based on an extended finite state machine model. Estelle is based on PASCAL, extended to describe finite state machines. All three of these languages describe the operation of the system in an algorithmic manner. LOTOS, based on process algebraic models and temporal reasoning, is superior in some cases for specifying standard network management systems because it is object-oriented[1]. The objective for these languages are virtually the same, to specify the interactions between protocol entities in a network. Their strengths and weaknesses are also similar, so we will only address LOTOS in detail.

LOTOS, an ISO Draft International Standard, is a formal description technique used for specifying the temporal ordering of events in a system. In this way, it is similar to Gist and PAISLey. Its use is primarily for describing protocols, not programs in general. It uses a notation similar to CSP[13] to describe the input and output behavior of processes. LOTOS can also be used to describe abstract data types, and the operations and transformations that data objects can undergo. LOTOS is designed to specify the messages exchanged by protocols. One can also use LOTOS to specify features of the OSI management framework, such as scoping, filtering, and access control. Because of these strengths, it has been proposed as a formal technique for specifying OSI network management entities, such as CMIS[24].

While LOTOS can specify parts of the network management system, it has some major gaps. Its support for the instantiation of data types and processes does not include explicit requirement of the locations at which processes are instantiated, nor does it include the ability to instantiate data objects. LOTOS also does not support

[1]The OSI network management architecture makes use of object-oriented paradigms[14,23,35] making LOTOS a good match. The SNMP architecture is not object-oriented[5,28], so LOTOS would not have any advantage here.

the possibility of timing or bandwidth constraints, although it does allow the specification of relative temporal ordering. Its access control function can be derived from the ability to hide CSP-style guards and encapsulation. With these, one can specify the access control policy desired, but one cannot specify how this policy relates to an instantiation of the processes in any particular environment. Our work concentrates on the possible interactions, and administrative and configuration relationships between management processes, not the low level protocol used to achieve those interactions.

These formal specification languages illustrate the current technological level for specifying programs. While we are not directly using any of these languages, because specifying networks and network management is different from the ways one would specify an algebraic program, we are incorporating specific capabilities of these languages into NMSL. Mechanisms for specifying process interaction are included in all of the languages mentioned above. The ability to specify timing constraints, as in PAISLey, is important. Separating the data abstractions from interactions, as in Larch, allows a cleaner, more easily understood specification. Executable specifications, as in PLEASE, help ensure that an implementation of the specification will conform to the specified parameters. These characteristics, in particular, are important to NMSL.

Chapter 3

Management Model and Overview

NMSL is designed to meet the high-level management needs of very large internets. It is designed to handle autonomous administrative domains and to provide facilities that allow cooperation between these domains. In this chapter, we first present our model for describing network management systems. Next, we outline properties of NMSL, a simple, formal language based on this management model. Third, we outline the consistency model used to evaluate NMSL specifications. Last, we give an overview of our tools for showing the correctness of these specifications, comparing specifications written by different administrators, installing these specifications, and enforcing them in a running network management system.

3.1. A Network Management System Model

NMSL is based on the model for network management systems shown in Figure 3.1. There are four main components to this model. Management data, ***objects***, are contained within a Management Information Base or

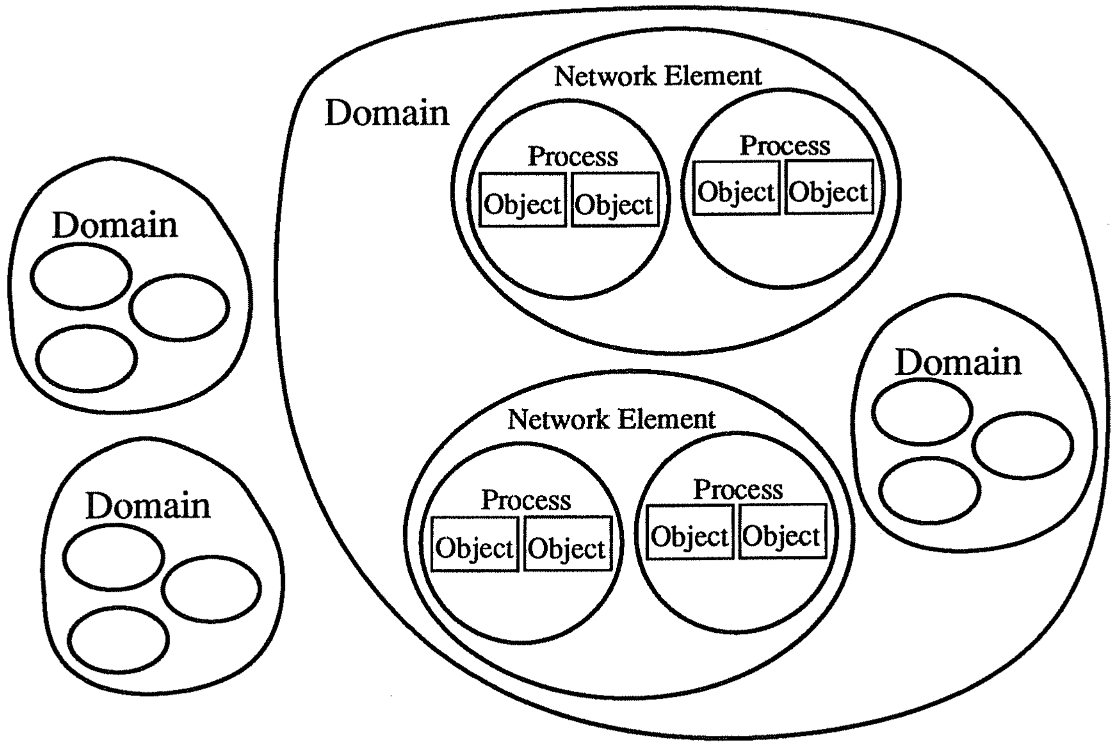

Figure 3.1.
The Management Model

MIB. This data is contained within *processes*. Processes communicate via queries to perform management functions. Management processes monitor and control the hardware and software elements of the system via queries on the MIB. Processes reside on the hardware or *network elements* (e.g. hosts, routers, bridges) attached to the network. These network elements are grouped into *administrative domains* (called simply *domains* in Figure 3.1).

Administrative domain descriptions group network elements, place restrictions on the interactions between domains, and place restrictions on the internal configuration of domains. Administrative domains can also contain other administrative domains; containment must be non-circular. Externally, an administrative domain should be considered a black box. Only those processes and queries that involve processes outside a domain are visible to other administrative domains.

3.2. A Formal Language

NMSL specifications are formal, allowing the use of automated proof techniques to find inconsistencies. NMSL specifications are also axiomatic, allowing us to specify the interactions (often called *queries*) between network management processes and their operation, without forcing a particular implementation. The axiomatic approach allows the specification of a generic management server or client application. The generic description need not depend on how the management function is implemented, as long as all implementations treat the management data in the same way and interact by making the same queries with the same frequencies.

The syntax of NMSL follows the system model closely. NMSL specifications are made up of the four components, objects, processes, network elements, and administrative domains. We describe each of these components in terms of the types of accesses that processes makes on objects, the rate at which accesses take place, and the constraints that the component makes on the configuration of the system.

Several characteristics are specified for each of the four components. An object description includes the name of the object, access permissions, and the relationship between this object and other objects (for example, objects could be organized hierarchically). A process description includes the queries that the process makes, and the exported queries other processes are allowed to make on this process. The specification of queries includes the objects being referenced and the type of reference operation. In addition, query specifications include timing characteristics, the frequency of requests, bandwidth requirements, and the speed at which servers can process requests. Network element descriptions describe the properties of the network element as they relate to network management, such as the CPU type, operating system, and network connections. Additional requirements for the configuration of the management system, such as the protocols used by processes, can be specified for both process and network element descriptions. Administrative domain descriptions group network elements and place restrictions of the interactions between processes by constraining access types and rates. Domain descriptions can also constrain the configuration of the network management system, such as the placement of processes or the types of network element allowed.

3.3. A Model for Consistency

The first step in using a NMSL specification is to determine if it is consistent. The model for consistency is made up of the three types of relationships present in NMSL: access types, access rates, and configuration constraints. Configuration constraints depend on the *context* of the constraint. The context of a constraint is the component of the system (object, process, network element or administrative domain) to which the constraint applies. The context can be the peer of communication (only applies to processes), the containing or instantiating component, or the component in which the constraint is specified. For example, a process may place a constraint on it's peer process in communication, the network element instantiating (containing) the process, or on itself. We say that a NMSL specification is consistent if and only if the following three conditions are met.

1) Every query access must be matched by an export permission. The export permission is given for each object, and can be restricted by either a process exporting the object or a domain containing the object.

2) The aggregate rate of all queries made on an object must be less than the exported rate at the desired loading level. The amount can be either the maximum load or the overload probability. In the maximum case, the exported rate can never be exceeded. In the overload probability case, the exported rate must not be exceeded the given percentage of the time, as specified by a client. The rate used, percentage or maximum, depends on the needs of the clients making the queries. Once again, the exported rate can be restricted by the containing domain.

3) Each configuration requirement must be matched by a configuration provision within the specified context.

NMSL specifications are automatically evaluated for consistency. The automatic evaluation is done by mapping the NMSL statements and consistency rules onto a Prolog-like language, CLP(R) [1]. Details about the model, this mapping, and consistency evaluation are given in Chapter 5.

3.4. NMSL Operation Overview

Figure 3.2 shows the parts of the NMSL system and their relationships. The boxes in the figure show the input and output of the NMSL system. The ovals represent active elements in the system. We divide the operation of this system into two aspects, the descriptive and the prescriptive aspects. The descriptive aspect includes the specification language, the NMSL Compiler and the Consistency Checker. The prescriptive aspect includes the NMSL configuration installers.

To use this system, administrators writes a specification for their network management systems. For simplicity, we will assume there is one administrator per administrative domain, and that each specification is written separately. This specification includes the properties outlined in the previous section. Included in this specification are the interactions both within the administrator's administrative domain and between this domain and other domains. The administrator compiles this specification and checks it for consistency. Any inconsistencies will be noted by the Consistency Checker. If the specification is consistent, the administrator can use the prescriptive aspect of the system to generate configuration output from the specification. The

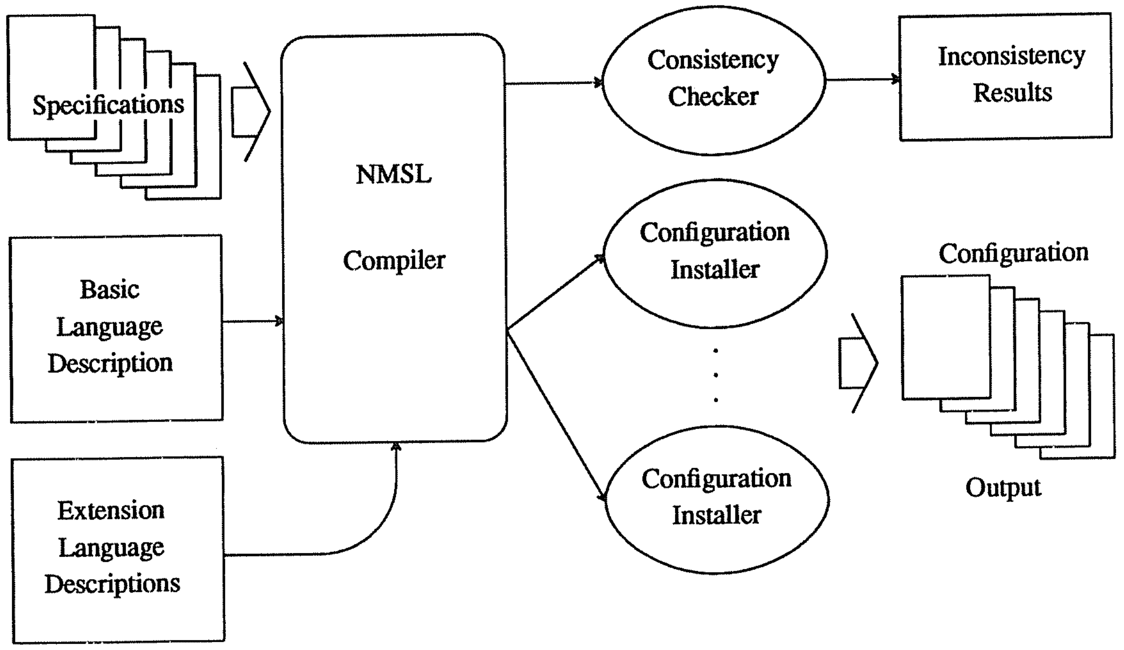

Figure 3.2.
The NMSL System Design

configuration output is made up of commands to reconfigure the network management system. New management processes can be started, existing ones can be reconfigured, and processes that are no longer needed can be terminated. After initial configuration, the system can continue to consult the specification and determine if the network is operating within the parameters of the specification. The specification can also be compared with new versions of the specification the next time the prescriptive step is performed.

3.4.1. Compilation

The compiler is central to both the descriptive and prescriptive aspects of NMSL operation. The compiler takes as input the network management specifications for each part of an internet. The output of the compiler varies depending on the desires of the system administrator. There are two basic types of output: consistency data, and configuration information. Thus, no matter what the administrator does with the specification, the first step is always to compile the specification into the appropriate form.

The syntax recognized by the compiler is fixed, but the semantic routines are external to the compiler and are read in at the beginning of compilation. The semantic routines are written in an extension language; they allow new specification types to be added to the basic NMSL repertoire. See Appendix D for an example use of the extension language. A system administrator or a vendor can use the extension language to write new specification types, relate them to the basic specifications, and describe the different types of compiler output in the extension language. Details of compiler operation are given in Chapter 4.

3.4.2. Consistency Checking

The consistency checker takes the consistency data from the specifications, adds some overall consistency requirements, and determines if the specifications for the network managers are consistent. If they are not consistent, the immediate causes for inconsistency are listed for the system administrator. Determining consistency is an important part of this research. The details of the consistency checker and the theory behind it are discussed in Chapter 5.

3.4.3. Configuration Installers

The configuration output is the prescriptive output of the NMSL system. The configuration installers take the configuration output from the NMSL compiler and install it, using a standard management protocol, in the network managers. Different network elements and processes require different configuration mechanisms, requiring multiple configuration installers. The installation of the specification-derived output is meant to guarantee that the configurations of the individual network managers meet the consistency requirements. To further ensure that the running system meets the specification, the running system can be periodically checked against the specification. If discrepancies are found, they can be reported or corrected. For example, if a process in the specification does not exist in the running system, it can automatically be restarted. Configuration installers are discussed in more detail in Chapter 6.

Chapter 4

The NMSL Language

This chapter describes the NMSL language. It describes the syntax and semantics of the language, the facilities the language provides, motivation for these basic facilities, and examples of each of the features. For a complete description of the NMSL language, see Appendix A.

4.1. Motivation

The purpose of a NMSL specification is to describe the expected interactions that take place in a system of network managers. In our model, an interaction is initiated by a client sending a request to a server. The server processes the request and returns a response. A request could be a simple look up operation, it could modify objects in the management database, or it could cause the execution of some remote operation on an object in the database. A process may act as a server in some interactions and as a client in others. To specify interactions of this form, we must specify the data that is transferred during the interaction, the parties involved, their expected behavior, where they are physically located in an internet, the administrative properties of the parties, and their groupings. We describe the behavior of the parties in terms of *protection*, *capacity* and *configuration* conditions. Protection conditions describe access permissions. Capacity conditions describe resource utilization. Configuration conditions describe requirements on the structure and implementation of the parties.

We first need to specify the types of objects in the network management database. The values of these objects will be transferred back and forth in queries. An object specification must include the abstractions present in the database: how the data is grouped into complex data types, and the various representations that an object can have. We call these *object descriptions.*

We also need to specify the clients and servers in the system; these are the parties involved in queries. These specifications are called *process descriptions.* A process description defines the queries that a process initiates, the frequency of the queries, and type of access (read or write). If the process is a server, the description also includes configuration information: which clients are allowed to query this server, how often, and for what data. A process description may also include statements defining how instances of object descriptions are transformed between their representations. Additional configuration requirements, such as protocols and hardware requirements can also be specified. A process in NMSL is an abstraction similar to an object description that must be instantiated at some place in the network.

The physical components of the network are specified in *network element descriptions.* A network element is any piece of hardware that can be connected to a network. Its description defines which processes are instantiated on it. It also specifies which parts of the management information base are supported (instantiated) on that network element.

Finally, the administrative relationships between groups of network elements and the processes that are instantiated on them are given in the *domain descriptions.* A domain description groups processes and network elements along administrative boundaries. Domains can overlap and nest. A domain description defines which network elements belong to a domain, which processes on those network elements can receive queries from outside the domain, and which processes are allowed to initiate queries to other domains.

The redundancy between the various descriptions is deliberate. Every network administrator has their own idea of how network elements, domains, and processes interact with other parts of the management system. To verify that all parts of the system are configured in a consistent way, the administrator needs to specify how each component interacts with other components, from that component's point of view. Multiple points of view cause the redundancy.

We separate of the specifications into abstractions (the data and process descriptions) their instantiations (on network elements) and administrative grouping (with administrative domains). This separation allows the management information to be specified independent of its use. It also allows types of processes to be specified, and allows these process types to be instantiated in various locations. These separate specifications, as shown in Larch[38], make them more generally useful. In the case of network management, the separation also mirrors the real world, where many network elements will store the same types of management data, and run network management software derived from the same source.

4.2. The NMSL grammar

The syntax of NMSL, is designed to by easy to write, understand and parse. Tokens are separated by white space or special character sequences like "::=" or ";". NMSL keywords are alphabetic. Various other token types are supported by NMSL, and their format is described when they are introduced. Tokens are grouped into sub-clauses, which in turn are grouped into clauses. Clauses are grouped into descriptions. There is a different type of description for each of the four components (type, process, network element, and administrative domain), as described in Section 3.1. Clauses are the basic descriptive element in the specification, and each clause describes a property of the component. The descriptions may occur in any order.

In this section, we describe the properties of NMSL. First, we present object descriptions. Next, we present process descriptions, and show their relationship to each other and to object descriptions. Third, we present network element descriptions. Last, we present administrative domain descriptions. Each description is accompanied by an example. In the examples, SNMP[5] and the IETF MIB[27] are used exclusively. For additional examples of the use of NMSL, see the full specification shown in Appendix B.

4.2.1. Object Descriptions

The NMSL syntax for object descriptions is an encapsulated version of ISO Abstract Syntax Notation One (ASN.1)[19]. ASN.1 has the advantages that it is general, machine architecture independent, and well known. It is used to specify the objects of both the IETF MIB and the OSI MIB (and the data formats used by other OSI protocols and standards), two of the management databases we might specify using NMSL. The `OBJECT-TYPE` and `OBJECT-CLASS` ASN.1 macro descriptions, used by the IETF standards[28, 35] are supported.

A NMSL object description is the keyword `module`, followed by a standard ASN.1 data module description, followed by a period. The full grammar and description for ASN.1 may be found in [19]. A module declaration ends with a period. Data may be imported from other type modules using the standard ASN.1 `IMPORTS` clause.

As an example of the use of an object description, consider the ASN.1 specification of the IP Address Table that defines the IP Addresses for a given entity. This example, shown in Figure 4.1, is derived from the TCP/IP MIB[27]. The `IpAddress` and `INTEGER` tokens are ASN.1 type names. `ipAdEntAddr`, et al, are members of the `IpAddrEntry` sequence. Complex types of this form are common in network management data.

```
module
    RFC1156-MIB
    { iso org(3) dod(6) internet(1) mgmt(2) 1 }

DEFINITIONS ::= BEGIN

IMPORTS
      mgmt, OBJECT-TYPE, IpAddress
          FROM RFC1155-SMI;

mib     OBJECT IDENTIFIER ::= { mgmt 1 }
ip      OBJECT IDENTIFIER ::= { mib 4 }

ipAddrTable OBJECT-TYPE
      SYNTAX  SEQUENCE OF IpAddrEntry
      ACCESS  read-only
      STATUS  mandatory
      ::= { ip 20 }

IpAddrEntry ::= SEQUENCE {
      ipAdEntAddr         IpAddress,
      ipAdEntIfIndex      INTEGER,
      ipAdEntNetMask      IpAddress,
      ipAdEntBcastAddr    INTEGER
}
END.
```

Figure 4.1.
Example Object Description

This simple specification shows how the ASN.1 type specification fits into a NMSL specification. It also shows how the access mode of a type applies to the components of the object; the access mode of `IpAddrEntry` was not specified, but `IpAddrEntry` was used in the definition of an `ipAddrTable`, so all `IpAddrEntry` elements in an `ipAddrTable` are read-only. The objects in the MIB are each given a unique identifier, specified as a list of numbers. So, for example, the identifier `{ ip 20 }` corresponds to the number list `1,3,6,1,2,1,1,4,20`. This list of numbers names the object and shows object containment (`1,3,...,4` corresponds to `ip`). Thus, `{ ip }` contains `{ ip 20 }`. The integration of the object descriptions with the other descriptions is given in the following sections.

4.2.2. Process Descriptions

A process description defines the characteristics of a network management process. A process is given a list of input parameters and performs some interactions with other processes based on the values of those parameters. A process description defines a new abstraction in the same way as an object description defines a new abstraction. Processes can be instantiated in a network element description.

A network management process can perform many functions, but we are mainly interested in its interactions with other processes and the network management data to which it has access. Therefore, in NMSL, one specifies the queries that a process makes, the data it makes available to other processes, and the configuration requirements of the process.

Figure 4.2 shows a simple example of a SNMP agent and a manager process.[2] The agent process, `snmpdReadOnly`, is a simple SNMP agent that exports all of its objects read-only to the ``wisc-cs'' domain. Because of the way the IETF MIB is defined, by supporting ``mgmt.mib'', the agent supports the full IETF MIB. It expects to be queried no more than 10 times per second without becoming overloaded. The process provides a protocol named ``snmp'' to its peer, and requires that its enclosing specification (the network element description) provide it with a protocol named ``udp''.

The manager process, `snmpPing`, retrieves the length of time an SNMP agent process has been running. This process resides on some instance-dependent network element. The manager has two parameters, `Proc` and `SysName`, which are assigned values when the manager is instantiated (executed). `Proc` is the name of the process to query, and `SysName` is the host on which the process runs. `SnmpPing` sends a query to `Proc` and requests the `sysUptime` value. This manager is expected to make queries infrequently, since it makes 1 query per second only 2% of the time (mode 1), the rest of the time the process is idle (mode 2). It requires that its peer can handle this load in the average case (this case corresponds to the `rate avg` in the specification), meaning that the destination should not be overloaded more than 50% of the time.

[2] An agent process stores management data and responds to requests while a manager process is one that initiates requests but does not store management data itself.

```
process "snmpdReadOnly" ::=
  supports mgmt.mib; -- entire MIB subtree

  exports to "wisc-cs" {
    objects mgmt.mib;
    access ReadOnly;
    rate max {
      mode 1 1.00 10 qps;
    }
  }
  queries "snmpt" {
    on "romano";
    traps any;
  }
  provides "protocol" = "snmp" to peer;
  requires "protocol" = "udp" of enclosing;
end process "snmpdReadOnly".

process "snmpPing" ( Proc, SysName ) ::=
  supports mgmt.mib; -- entire MIB subtree
  queries Proc {
    on SysName;
    requests mgmt.mib.system.sysUpTime;
    rate avg {
      mode 1 0.02 1 qps;
      mode 2 0.98 0 qps;
    }
  }
end process "snmpPing".
```

Figure 4.2.
Example Process Descriptions

This example shows the main points of the `process` descriptions. The objects supported by the process are listed in the `supports` clause. Each type listed must be defined in an object description. The interactions are specified using the `queries` and `exports` clauses. The types listed here must be listed in an object description, and a `supports` clause. `Queries` clauses describe the active side of communication, `exports` clauses describes the passive export of data to other processes or domains. A `queries` clause specifies the process being queries, the network element on which the process runs (the `on` subclause), parameters to the query (the `using` subclause, not shown in the example), the expected response (the `requests` subclause), the data modified (the `modifies` subclause, not shown in the example), possible traps or exceptional alarms to be sent (the `traps` subclause), and the timing characteristics of the query (the `rate` subclause). Note that in this example, the process sends `traps` to a process named `snmpt` instantiated on the system named `romano`. Any trap allowed by the protocol can be sent, and the rate is unspecified (meaning the rate is probably to low to be significant).

The `rate` subclause requires additional explanation. This subclause expresses the rate at which the process sends or receives queries, and, in the case of a `queries` clause, specifies requirements on the rate of the peer. The activity that we are modeling is that of a typical management process, where the process has different query behaviors throughout its lifetime. For example, a process might have 3 different modes of behavior. In the first mode, the process makes a complete dump of router statistics twice a day. In the second mode, the process gets updates from each router on the hour. The remaining time the process is idle, making no queries. Each of these modes is associated with a different level of activity. Each mode is characterized by a percentage of time spent in that mode and the associated level of activity. The percentage is the amount of time over the lifetime of the process that it executes in the mode.

In a `queries` clause, we can specify three types of rate requirements, `max`, `avg` or a percentage. These are requirements on the exporter of the data. If `max` is specified, this process requires that the recipient of the query never be overloaded, no matter what combination of queries and modes are made. If `avg` is specified, this process requires only that the recipient of the query not be overloaded in the average case, where average means half of the time. `Avg` is a special case of the percentage case. If a percentage is specified, then the recipient of the query must be able to handle the query the given percentage of time, without causing overload. For example, if 75% is specified, then the recipient must be able to process queries 75% of the time without becoming overloaded. The way in which overload is calculated is described in Chapter 5.

The `provides` and `requires` clauses describe other constraints on the configuration of the system. A `provides` clause specifies a ***configuration condition*** and a value or range of values associated with that condition. A configuration condition can be tested for existence, and instances of condition conditions can be counted. The range associated with configuration conditions can also be tested. A `requires` clause makes a requirement on a configuration condition provided by a `provides` clause. Specifying a range in a `requires` clause says that all values of the configuration condition given in a `provides` clauses must be in the specified range. `Provides` and `requires` clauses can also be specified in network element and domain descriptions. Additional details of these clauses are described in examples later in this chapter and in Chapter 5.

A `provides` or `requires` clause applies only to the ***context*** specified. The context of the provision or requirement is listed using the `to` or `of` subclauses. If the context is `peer`, the condition applies to the peer of communication. If the context is `containing`, the condition applies to the containing context. A network element contains processes, and a domain contains domains and network elements. For example, specifying `containing` within a process description means the condition applies to all network elements that instantiate the process. If the context is `self`, the condition applies to the current description and any contexts listed within the description. For example, specifying `self` within a process description means the condition applies to all clauses of the process description.

4.2.3. Network Element Descriptions

Network element descriptions describe the physical properties of the network. Each description shows the details for a single computer, terminal, printer, router or other device that could be connected to a network, and its connections to an internet. It also lists the portion of the MIB that is supported by the network element's hardware and operating system. Finally, a list of the network management processes that are expected to execute on this network element is given.

An example of a network element description is shown in Figure 4.3. This network element, called `romano.cs.wisc.edu`, has a single interface, connecting it to a 10Mbps ethernet named `wisc-research`. It runs the Ultrix version 4.0 operating system. It supports the objects listed from the IETF MIB for SNMP[3]. `Romano.cs.wisc.edu` instantiates two process, one called snmpt (an SNMP trap handler) and one called snmpdNoEgpReadOnly (like the snmpReadOnly process shown in Figure 4.2, but without EGP support). It also sets some internal objects, shown by the `provides` clauses.

Several important features of network element descriptions are shown in this example. The properties of the hardware are given, including the CPU type and a list of network interfaces. Each `interface` lists one network interface, an identifier, the physical network to which the interface connects, the type of interface, and its

```
system "romano.cs.wisc.edu" ::=
  cpu "decstation";
  interface "ln0" {
    net "wisc-research";
    type "ethernet-csmacd";
    speed 10 Mbps;
  }
  opsys "ultrix" version "4.0";
  supports
    mgmt.mib.system, mgmt.mib.at,
    mgmt.mib.interfaces,
    mgmt.mib.ip, mgmt.mib.icmp,
    mgmt.mib.tcp, mgmt.mib.udp;
  process "snmpt";
  process "snmpdNoEgpReadOnly";
  provides "sysContact" =
      "Dave Cohrs <dave@cs.wisc.edu>";
  provides "sysLocation" = "6372 Comp Sci";
  provides "protocol" = "udp";
end system "romano.cs.wisc.edu".
```

Figure 4.3.
Example Network Element Description

[3]A reader familiar with the IETF TCP/IP MIB will notice that the EGP objects are not supported.

nominal speed. The speed is important for determining if the processes on this network element will be able to respond to queries in a timely manner, or if this network element will be swamped with management requests. The software is also described. The operating system provided is listed (specific operating systems or versions could be required by different processes). The MIB objects that the network element supports are also listed. Each network management process that is instantiated on this network element is listed in the `process` clauses. Parameters may be listed for the process instantiations (not shown in this example).

4.2.4. Domain Descriptions

Domain descriptions describe the administrative groupings found in networks. Domains may contain other domains (sub-domains) and network elements. A domain description also lists how its members relate administratively to other domains.

An example of a domain description is given in Figure 4.4. In this example, the domain, `wisc-cs`, contains two network elements. The `wisc-cs` domain exports the full IETF MIB to the `public` domain, but only for reading, and requests from `public` can arrive no more frequently than once per second without causing overload.

This example shows the main feature of domain descriptions, grouping network elements into administrative domains. Subdomains can also be grouped within domains. The `exports` clause specifies which parts of the MIB that are available in this domain can be accessed by processes in other domains. The `exports` clause is allows the enforcement of higher level limitations on lower level exports. For example, an administrator of a host or subdomain may allow data to be exported to any other domain. The administrator for the containing domain can restrict this exported data, by restricting both the permissions granted or the rates available to non-local domains. One can also specify `provides` and `requires` clauses, just as for process and network element descriptions. They can be used to restrict interactions between domains or to restrict configurations within the domain itself. In the example, we require that there be at least 1 instance of the `Router` configuration

```
domain "wisc-cs" ::=
  system "romano.cs.wisc.edu";
  system "asiago.cs.wisc.edu";
  exports to "public" {
    objects mgmt.mib;
    access ReadOnly;
    rate max { mode 1 1.00 1 qps; }
  }
  requires count("Router") >= 1 of self;
end domain "wisc-cs".
```

Figure 4.4.
Example Domain Description

condition within the domain. This shows another feature of the `requires` clause, the ability to count the number of configuration condition instances within a context, rather than just comparing ranges.

4.3. Summary

The examples given above, taken from a description for a management system set up at the University of Wisconsin Computer Sciences Department, show the descriptive ability of NMSL. The language can specify the management objects using the standard ASN.1 notation. It also specifies the operation of the management system via the process descriptions. The process descriptions describe not only the data accessed and exported, but the operations performed on the data, and the frequency at which the queries are made. Furthermore, processes can be parameterized. Network element descriptions describe the hardware and the software components of the network element, as they affect network management. In addition, processes can be specified for each network element, and the same process type to be used on multiple network elements. Finally, domain descriptions group network elements, and can restrict the interactions between domains and restrict the configuration of the domains.

Chapter 5

Consistency Evaluation

Given a network management specification, the NMSL system must inform the system administrator if the specification is consistent. If it is not, the causes for inconsistency should be shown to the administrator. This is the task for the NMSL Consistency Checker. The consistency checker is based on a network management specification consistency model.

There are two basic ways to evaluate consistency, centrally and incrementally. We first investigated a centralized approach to evaluate specifications. In this approach, all specifications are copied to a single location and evaluated. Each of the possible relationships is tested, and if all relationships meet the model's conditions, the specification is consistent. This approach works well for small specifications of single administrative domains.

For multiple domains, the centralized approach violates the goal of preserving the autonomy of different administrative domains. By copying all the data to a central location, administrators give up information that may be proprietary and may not be needed by any specification other than the specification for their own domains. Furthermore, the centralized model does not scale well. The time it takes to evaluate specifications increases non-linearly. Justification for this claim is given in Chapter 7. To address both of these problems, we investigated incremental evaluation methods.

In an incremental method, one divides the set of specifications into pieces and evaluates each piece separately. The conditions for consistency remain the same as in the centralized method. Because it may not be possible to break up the specification into independent (non-interacting) pieces, relationships between the pieces must be determined and evaluated incrementally. As in the centralized approach, if all the relationships meet the consistency conditions, the specification is consistent.

In the incremental method that we investigated, we divided evaluation along domain boundaries, taking advantage of the hierarchical organization of administrative domains for evaluation and incremental re-evaluation. In the first step, the specification of each bottom level domain is checked for consistency using the centralized method. Then, relationships between specifications are determined. The inter-domain relationships are sent to the immediate parent in the hierarchy. When the parent has been notified of all the relationships from its children, it evaluates these relationships for consistency along with its own specification, once again, using the centralized method. If any inter-domain relationships remain (with domains from elsewhere in the hierarchy), they are sent to the parent domain for evaluation. This process repeats until the top is reached, when a final evaluation takes place. If the evaluation succeeds, all of the specifications are consistent, and all domains are notified. Otherwise, at any point inconsistencies are determined, they are reported back to the child domains.

The same hierarchical method is applied when a specification is modified. In this case, re-evaluation can stop at any point in the domain tree where the set of inter-domain relationships remains unchanged from the previous consistency check. In other words, if only the internal part of a domain's specification changes, the consistency check need only be performed on that domain's specification (along with its children's inter-domain summaries).

In this chapter, we first describe the model for consistency used for centralized evaluation. Next, we show how the model is expressed in an underlying logic, CLP(R). Third, we extend the evaluation model to allow incremental evaluation and to take advantage of the distributed nature of network management and administrative domains. We have implemented an incremental evaluator, using the methods described in this chapter. Experience with this evaluator is presented in Chapter 7.

5.1. The Centralized Consistency Model

The consistency model states the conditions that must be met for a given set of specifications to be correct. The consistency model assumes a *client/server*[4] based system. We divide the conditions for consistency into the three NMSL categories: protection, capacity, and configuration. These categories correspond to the three types of relationships specified in a NMSL specification. Any given specification is defined in terms of these three relationships. The protection condition states that a client must be given permission to perform the requests that it makes. The capacity condition states that a service provider must have enough capacity to handle the requests from all of its clients. 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.

In this section, we develop a logical semantics for NMSL. The goal of developing these semantics is to logically define inconsistency. Based on the three conditions for inconsistency, we can state in logic the condition for an inconsistent specification:

$$\begin{array}{l} \mathit{ProtInconsistent} \mid \mathit{CapacityInconsistent} \mid \\ \mathit{ConfigInconsistent} \mid \mathit{CountInconsistent} \supset \mathit{Inconsistent} \end{array}$$

This condition states that if there is a protection, capacity, or configuration inconsistency, then the specification is inconsistent. ***ProtInconsistent*** and ***CapacityInconsistent*** correspond to inconsistent protection and capacity conditions, respectively. ***ConfigInconsistent*** and ***CountInconsistent*** correspond to inconsistent configuration conditions. In sections 5.1.1 through 5.1.3, we describe each of these consistency categories in more detail.

[4]Network management standards[5, 14] refer to these processes as ***managers*** and ***agents***, respectively. We prefer to use the more general, distributed systems terms.

5.1.1. Protection

The purpose of the protection conditions is to verify that all operations that a client requests of a 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 conditions include the property that each request performs a single operation on a single object.

To verify the protection condition, we must show that each client has permission to perform each of its specified queries. In this discussion, we use 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 network element or domain D_2.
$Instan_D\ (D, P, I_{D,P})$	Network element 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 $I_{P,O}$.
$Ref\,(C, I_S, T, I_{S,O})$	An instantiated process, C, requests to perform operation T on object instance $I_{S,O}$ from process instance I_S.
$Perm\ (C, I_S, T, I_{S,O})$	Domain or process instance I_S permits domain or process instance 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 network elements and sub-domains 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. *Ref* 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. The functions described above are related by a small set of deduction rules, shown in Figure 5.1.

The notation used is first order logic. The = relation denotes equality, which is true where the values compared are equal. Universal quantification is assumed for all variables listed in the rules.

The Transitivity rules formalize the intuitive notion that an enclosing domain or object contains all of the parts of its subdomains or sub-objects, respectively. 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 network element instantiates a process, the network element 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.

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

Figure 5.1.
Logical Deduction Rules for Protection Verification

The Implied Object Permission rule states that if a process gives permission to access an object instance, and the object contains sub-objects, then the process also gives permission to access the instantiations of those sub-objects.

The Implied Domain Permission rule states that if a process or a domain gives some access permission to a domain, then anything contained in that domain also has that permission.

The rules for defining Unique Identifiers state that instance identifiers with respect to protection 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 can prove that a specification is consistent if, for every request, there exists a corresponding permission. Conversely, we say that a specification is inconsistent if there is any reference to an object for which permission is not granted. This can be expressed logically as

$$\begin{aligned}(\exists\ C, I_S, T, I_{S,O} :\ & \mathit{Ref}(C, I_S, T, I_{S,O})\ \& \\ & \neg\, \mathit{Perm}\ (C, I_D, T, I_{S,O})\ \& \\ & (I_S = I_D \mid (\mathit{Contains}_D\ (I_D, I_S)\ \&\ \neg\, \mathit{Contains}_D\ (I_D, C\ ))) \\ \supset \mathit{ProtInconsistent}. & \end{aligned}$$

This says that a specification is inconsistent if there is a query from process instance C of type T to object $I_{S,O}$ at destination I_S, for which permission is not granted by I_S or any domain I_D that contains I_S and does not contain C. This condition allows members of the same domain to access data that is not accessible from outside the domain. This condition also allows members of other domains to be constrained by domain-wide permission restrictions.

The evaluation is simplified by the finite problem space – we only need to prove this condition for the statements in a specification. A simple, centralized method proceeds as follows. We attempt to match each *Ref* relation with its corresponding *Perm* relation. The *Perm* relations are either taken directly from a translated NMSL specification, or are derived. We employ the Implied Object Permission rule to find the permissions for each object instance, using Instan_P, and $\mathit{Contains}_O$ relations obtained from the specification. Next, we use the derived *Perm* relations, along with the ones obtained directly from the specification, the $\mathit{Contains}_D$ relation, and the Implied Domain Permission rule to derive the permissions for each client process and the containing domains. If, after iterating through the *Perm* relationships, any *Ref* relations have no match, the specification is invalid.

5.1.2. Capacity

The goal of the capacity conditions are to determine, as quickly as possible, if each server has the capacity to provide the services needed specified by its clients. This is a form of the classic capacity planning problem[4].

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 of a system, and the 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.

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 that 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 that must be calculated quickly.

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. This assumption limits the interactions that can be accurately modeled. We make this assumption to avoid the use of queuing models. We plan to incorporate a more complex interaction model in the future.

Capacities are calculated by using an independent, discrete distribution of the frequency of interactions (requests and responses), between network management processes. A client's behavior is divided unto distinct modes of operation. If we inspect the distribution of queries of the client over a sufficiently large period of time, we can divide this distribution into modes, where the total percentage of time spent in each mode is the amount shown in the specification. The order of the modes over time is random. A client can be in the same mode more than once over the time interval measured, as long as the total percentage is the amount specified. During a mode of operation, a client makes at most q queries per any 1 second interval, or 1 query per $\frac{1}{q}$ seconds.

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. These loads are calculated from a discrete aggregate distribution of the group. Interactions are independent, and the order of modes of the individual clients is random. Therefore, the aggregate distribution has several modes, corresponding each combination of modes of the independent clients. When measuring a sufficiently large period of time, the total percentage of time spent in each mode is the aggregate percentage. As in the case of one client, during a mode of operation, the aggregate of clients makes at most q queries per any one second interval.

Several concepts are not present in our model for simplicity's sake. Interactions are independent, so there are no interactions between request by the same client, nor are there interactions between clients querying the same server. 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 or processing 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. Response time could be included by using a queuing model, however given the large number of states in the model, we would not have the resources (time or memory) to solve the problem, given current performance modeling technology[25]. Finally, we also do not model the fine-grained operation of the message delivery protocols.

We compare the capacity, in queries per second, of servers, with the load clients will place on those servers, and determine whether each server's capacity will be exceeded. The load can be measured several ways. We are interested in the *peak utilization*, the maximum number of queries a server can receive and process per second. We are also interested in the probability that a server's capacity will be exceeded, called the *overload probability*. 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.[5] 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 interesting from a reliability point of view[3], 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 5.2. The peak utilization is 20 *qps*, and the aggregate distribution is the same as the simple distribution. 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 5.2, the server, which can receive 20 *qps*, has the capacity for the peak rate of requests, and any percentage less than the peak. Therefore, the probability that the server's capacity will be exceeded is zero.

In the multiple client/single server case, we simply add the peak modes of the individual clients to get peak request frequency. For example, Figure 5.3 shows three clients $Client_1$, $Client_2$, and $Client_3$. The peak request rate is the sum of the peak rates for each of the clients,

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

which is greater than the capacity of the server.

We calculate the overload probability by computing an aggregate distribution. The distribution of the aggregate load of these three clients has 8 modes, and is determined using simple probabilities, because of our independence assumption. 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

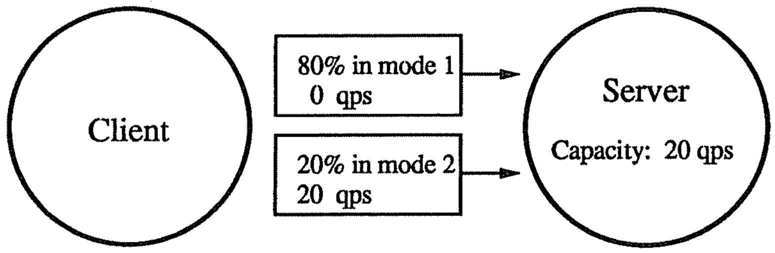

Figure 5.2.
A Single Client/Single Server Configuration

[5]The examples in this section all show clients with two modes of operation. The query model is not limited to two mode distributions; any number of modes can be specified.

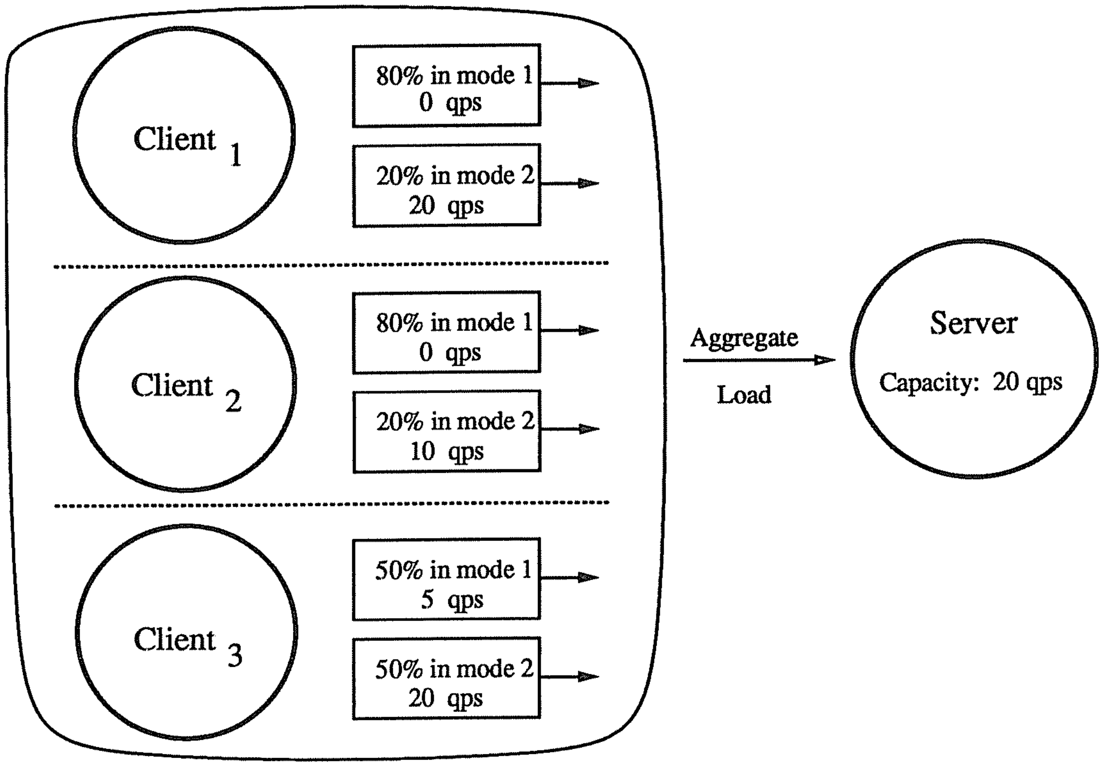

Figure 5.3.
A Multi-client/Single Server Configuration

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

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

The other modes are calculated in the same manner. The resulting aggregate distribution, shown in Figure 5.4, is

Mode	Prob.	Query Rate	Mode	Prob.	Query Rate
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 5.4).

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 peak aggregate load is

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

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 a 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 $O\ (m^n)$.

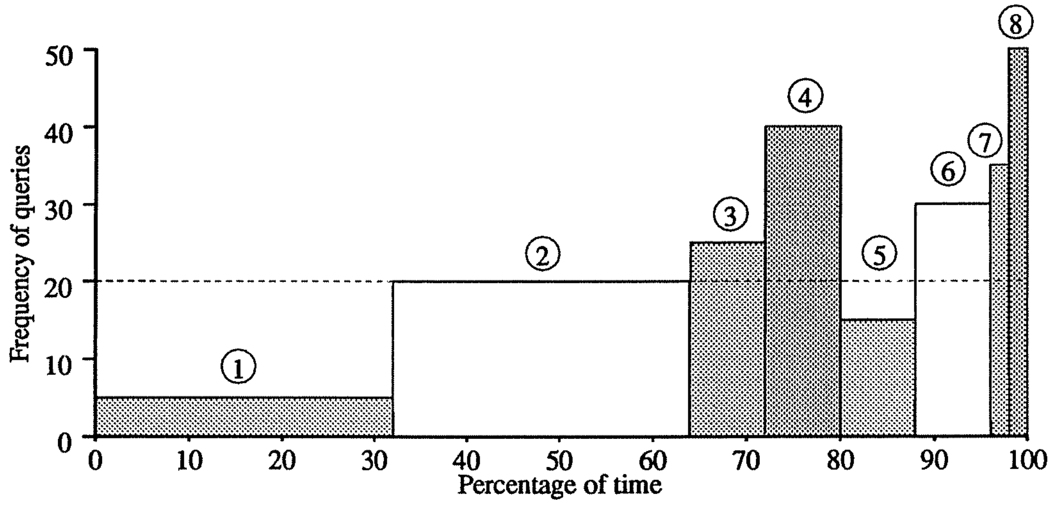

Figure 5.4.
Aggregate Load Distribution of Three Clients

We can limit the number of modes in the output 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. 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 maximum of the frequencies of the calculated modes.

To determine the amount of error introduced by approximation, we can measure the amount of area added to the distribution graph. To limit the error, we make two restrictions on the combination of modes. First, that we order the pre-combined modes according to their frequency (height). We combine modes in order until the total percentage in the combined mode is greater or equal to $\frac{1}{N}$, and that the sum of the remaining percentage is $\frac{1}{N}$ times the number of remaining combined modes. The percentage is the width of the mode, as shown in Figure 5.4.

By picking the frequency of the combined mode to be the maximum of the modes in the group, we guarantee that the calculations will never determine that a group of clients is within a server's capacity, when the capacity is exceeded. This also corresponds with our model, where, if we take an aggregate distribution mode the number of queries for any second interval is bounded by q. Logically, however, there is a chance of an inconsistency being determined when a server does have enough capacity. This chance of error decreases as N increases.

If we take the aggregate load shown in Figure 5.4 and combine them into 4 modes this way, we get the following modes:

Mode	Old Modes	Probability	Query Rate
Mode 1	1	.32	5 *qps*
Mode 2	5,2	.40	20 *qps*
Mode 3	3,6	.16	30 *qps*
Mode 4	4,7,8	.12	50 *qps*

The total amount added to the distribution is calculated by adding up the area under the distribution graph and subtracted from the area of the original graph. The area of the original distribution is 18.5*qps*. The area of the approximate distribution is 20.4*qps*. The difference is 1.9*qps*, for an error of about 10%.

We can calculate this approximation for N modes in polynomial time, using a pairwise aggregation method. In this method, we take pairs of clients and calculate their combined aggregate. Since each client has at most N modes, each pair requires $O\ (N^2)$ calculations. From this result, we choose N modes, as discussed above. We repeat this process, tournament style, until all pairs have been combined. There are $n-1$ total aggregations. This results in $O\ (N^2 \times n)$ calculations for the entire aggregate. Remember that N is constant for any given consistency check, so the time varies linearly with the number of clients in the aggregate.

In determining consistency, we need to prove that all client capacity requirements are matched by large enough capacity exports for the client's needs. Conversely, there is an inconsistency if there is any client capacity requirement which cannot be satisfied. To express this logically, we introduce a relation, *AggrTooBig* to represent the calculations shown above,

$$\exists\, AggrTooBig\ (C, I_{s,o}, T_C, T_{s,o}) \supset CapacityInconsistent.$$

This relation states that there is an inconsistency if the aggregate is too large. The aggregate is where process C is making queries of object instance $I_{s,o}$ with aggregate rate T_C (this rate takes into account the client requirement of overload probability or peak load), but the exported rate was $T_{s,o}$.

5.1.3. Configuration

The configuration conditions are used to verify parts of NMSL specifications relating to the configuration of network management systems. This classification includes all parts of a specification not modeled by the capacity or protection conditions. 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: the protocol used to communicate between two management processes is the same, the maximum number of routers allowed in a network is less than 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 5.5. These examples are fragments of a NMSL specification (other configuration conditions are shown in the examples in Chapter 4).

```
system "romano.cs.wisc.edu" ::=
  cpu "decstation";
  interface "ln0" {
    net "wisc-research";
    type "ethernet-csmacd";
    speed 10 Mbps;
  }
  opsys "ultrix" version "4.0";
  provides "sysContact" =
          "Dave Cohrs <dave@cs.wisc.edu>";
end system "romano.cs.wisc.edu".

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

Figure 5.5.
Examples of Configuration Conditions

The first example shows the hardware configuration of a computer on our local network, defining several 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 `provides` clause defines a name that is used in configuring the network element.

The second example specifies, in a `requires` clause, the requirement that the number of computers that route packets between networks (i.e. those computers that run routing processes) be between 1 and 3, inclusive. Requirements can also be used to enforce administrative needs. Specifying a minimum of one router, for example, enforces the requirement that a network cannot be split into two disconnected networks without changing the specification.

To verify configuration conditions, we must show that each condition in a specification is met within the specified context. The context specifies the scope of the requirement (self, containing, or peer), as described in Section 4.2 for the `provides` and `requires` clauses. A configuration condition is a name; the number of instances of this name in a context can be counted, and the range or ranges of values associated with these instances can be tested.

To verify conditions concerning a count of a condition, we determine the number of instances of the condition in the specification within the specified context. The method used for verifying range conditions is similar to the method used for verifying protection conditions. The major difference between the methods used for determining configuration conditions from protection or capacity conditions is the addition of the context.

The following logical relations are used to represent configuration conditions:

Provides (D_1, N, OP, V, D_2)

Context D_1 (a process, network element or domain) provides condition N to Context D_2. OP and V define the value or values provided. OP is one of $=$, $<$, $\leq$, $\geq$, or $>$. For example, *Provides* $(D_1$, *"protocol"*, "=", *"udp"*, $D_2)$ means that context D_1 provides protocol *"udp"* to context D_2.

Requires (D_1, N, OP, V, D_2)

Context D_1 requires condition N of Context D_2. OP and V define the value or values required. Their values are of the same form as the *Provides* relation.

Count (D_1, N, OP, V, D_2)

Context D_1 requires a count of condition N in Context D_2. OP and V define the value or values required. OP is one of $=$, $<$, $\leq$, $\geq$, or $>$. V must be a number. For example, *Count*$(D_1$, *"Router"*, "<", *4*, $D_2)$ requires the count of *"Router"* instances provided by context D_2 to be less than 4.

D_2 depends on the context specified. If the context is the peer, D_2 can be any context that queries or is queried by D_1. If the context is the containing context, D_2 is any context that contains or instantiates D_1. Otherwise, the context is the same as D_1.

As in the case of protection conditions, we need a small set of rules to evaluate conditions of instantiated processes (shown in Figure 5.6). The first rule states that if context D_1 instantiates process P with identifier I, and P provides condition N with value range OP,V to context D_2, then all instances I also provide N with the given range to D_2. The other two rules define the same relationship for *Requires* and *Count* relations.

For each instance of a *Requires* relation, we search for a matching *Provides* relation. The *Provides* relations are either taken directly from a translated NMSL specification or are derived using a Implied Instantiation rule. If any *Requires* relation does not have a corresponding *Provides*, the specification is invalid. This is expressed in logic as

$$(\exists\, D_1, N, OP, V, D_2 : \mathit{Requires}\,(D_1, N, OP, V, D_2)\ \&\ \neg\, \mathit{Provides}\,(D_1, N, OP, V, D_2)) \supset \mathit{ConfigInconsistent}.$$

For evaluating counts, we make an aggregate of all instances of condition N provided in the specified context. If the aggregate is within the bounds specified by OP,V, the condition is valid, otherwise it is invalid. We express this logically with the implication

$$\exists\ \mathit{BadCountRange}\,(D_1, D_2, N, OP_1, C_1, C_2) \supset \mathit{CountInconsistent}.$$

BadCountRange represents the meta-logic operation of counting the *Provides* relations and comparing this count with specified range. This relation states that there is an inconsistency if context D_1 requires context D_2 to provide instances of condition N in the range OP_1, C_1, but there are C_2 instances, and this is not in the range. For example, if we prove that

$$\exists\, \mathit{BadCountRange}\,(D_1, D_2, \text{"Router"}, \text{"<"}, 4, 6)$$

then D_1 required that D_2 provide less than 4 router instances, but D_2 provides 6.

Implied Instantiation	$\mathit{Instan_D}(D_1, P, I)\ \&\ \mathit{Provides}\,(P, N, OP, V, D_2) \supset \mathit{Provides}(I, N, OP, V, D_2)$
	$\mathit{Instan_D}(D_1, P, I)\ \&\ \mathit{Requires}\,(P, N, OP, V, D_2) \supset \mathit{Requires}(I, N, OP, V, D_2)$
	$\mathit{Instan_D}(D_1, P, I)\ \&\ \mathit{Count}\,(P, N, OP, V, D_2) \supset \mathit{Count}(I, N, OP, V, D_2)$

Figure 5.6.
Logical Deduction Rules for Configuration Verification

5.2. Consistency Evaluation in CLP(R)

To evaluate NMSL specifications, we translate the specification into the logic programming language, CLP(R). CLP(R) is similar to Prolog, but allows reasoning about relationships over the real numbers. We show that statements of NMSL can be translated into CLP(R) and that CLP(R) is powerful enough to represent the consistency model. In doing so, we show how the logic rules presented in Section 5.1 are represented in CLP(R). In this section, we use examples of NMSL statements and their translations. These examples are taken from a full specification for a network management system. The full specification and translated CLP(R) code are given in Appendix B. The full translation of the evaluation rules is shown in Appendix C.

Each statement of NMSL translates into a CLP(R) clause corresponding to one of the relationships discussed in Section 5.1. For each object listed in a `module` declaration, we generate clauses of CLP(R) defining the object containment and protection. For example, within the TCP/IP MIB,

```
sysDescr OBJECT-TYPE
      SYNTAX  OCTET STRING
      ACCESS  read-only
      STATUS  mandatory
      ::= { system 1 }
```

is represented by

```
contains([[1,3,6,1,2,1,1],[1,3,6,1,2,1,1,1]]).
decl_access([1,3,6,1,2,1,1,1],readOnly).
```

The strings of digits are the numeric identifiers for the names `system` and `sysDescr` respectively. `Decl_access` is part of the representation of the *Perm* relation.

Each `export` access permission type is represented by a clause describing the access type and relating that type to every instance of the export. The rate modes are represented by similar clauses. For example,

```
exports to "wisc-cs" {
  objects mgmt.mib;
  access ReadOnly;
  rate max {
    mode 1 1.00 10 qps;
  }
}
```

is represented by

```
perm_access(S2,'wisc-cs',Y,readOnly) :-
        instanD(S1,process('snmpdReadOnly'),S2),
        instan(S2,[1,3,6,1,2,1 | V1],Y).
perm_eq('wisc-cs',Y,1,10) :-
        instanD(S1,process('snmpdReadOnly'),S2),
        instan(S2,[1,3,6,1,2,1 | V1],Y).
instan(P1,[1,3,6,1,2,1],[P1,1,3,6,1,2,1]) :-
        instanD(S2,process('snmpdReadOnly'),P1).
```

`Perm_access` corresponds to the logical *Perm* relation; `perm_eq` corresponds to a rate mode. The value `1,3,6,1,2,1` is the numeric form of the data exported, `mgmt.mib`. The unique identifiers, `S2` and `Y` are derived from clauses output from the network element descriptions and from the `instan` clause shown at the end of the example. The `instan` clause describes all of the instances of objects supported by this process.

The `queries` clause is represented similarly. In addition to protection and capacity rules, we generate a special rule used for representing peer relationships for `provides` and `requires` statements. For example, the statement for a process named `snmpxmon`,

```
queries "snmpdReadOnly" {
  on gw;
  requests
    mgmt.mib.interface.ifTable.ifEntry.ifOperStatus;
  rate avg {
    mode 2 0.02 1 qps;
    mode 1 0.98 0 qps;
  }
}
```

is represented by

```
ref_access(P2,Y1,readOnly) :-
        instanD(S2,process('snmpxmon'),P2),
        instanD('gw',process('snmpdReadOnly'),P1),
        (instan(P1,[1,3,6,1,2,1,2,2,1,8],Y1)).
ref_eq(P2,Y1,0.02,1,avg) :-
        instanD(S2,process('snmpxmon'),P2),
        instanD('gw',process('snmpdReadOnly'),P1),
        (instan(P1,[1,3,6,1,2,1,2,2,1,8],Y1)).
ref_eq(P2,Y1,0.98,0,avg) :-
        instanD(S2,process('snmpxmon'),P2),
        instanD('gw',process('snmpdReadOnly'),P1),
        (instan(P1,[1,3,6,1,2,1,2,2,1,8],Y1)).
queries(P1,P2) :-
        instanD(S1,process('snmpxmon'),P1),
        instanD('gw',process('snmpdReadOnly'),P2).
```

The `ref_access` clause corresponds to the *Ref* relation. The `ref_eq` clauses correspond to the rate modes. Once again, the strings of digits correspond to the objects being referenced. The `queries` clause states the process instances potentially involved in peer relationships.

The configuration requirements (from the `provides`, `requires`, and network element configuration clauses) are also represented by clauses. For example, in a process declaration for the process `snmpdReadOnly`,

```
provides "protocol" = "snmp" to peer;
requires "protocol" = "udp" of enclosing;
```

is represented by

```
provides(Y2, 'protocol', '=', 'snmp', X) :-
        instanD(S1,process('snmpdReadOnly'),Y2),
        queries(X,Y2).
requires(Y2, 'protocol', '=', 'udp', X) :-
        instanD(S1,process('snmpdReadOnly'),X),
        Y2 = S1.
```

Note the use of the `queries` clause for determining all peer instances in the `provides` clause. Recall that the enclosing declaration for a process is a network element declaration. Since network elements instantiate processes, all of the enclosing instances are determined using the `instanD` relationship. `Provides` and `requires` clauses in network element and domain descriptions are represented similarly.

We also generate CLP(R) clauses describing the instantiation of processes on network elements. For example, the statement within the network element description for `romano`,

```
process snmpdNoEgpReadOnly;
```

is represented by

```
instanD('romano',process('snmpdNoEgpReadOnly'),
        [romano,snmpdNoEgpReadOnly]).
```

Finally, domain grouping is represented by clauses describing containment. For example, in the domain `wisc-cs`

```
system romano;
```

is represented by

```
containsD(['wisc-cs','romano']).
```

Representing our consistency conditions in CLP(R) is straightforward. The rules for Transitivity, Implied Containment, Implied Instantiation, Implied Object Permission and Implied Domain Permission, shown in Figure 5.1, are already in clausal form and translate directly into CLP(R). The Unique Identifier Rules are incorporated into the Implied Instantiation Rule and the relationships output by the NMSL compiler. That is, the compiler generates unique identifiers, such as the one shown in the process instantiation example above. The translation of other rules are shown in Appendix C.

The protection condition shown in logic in Section 5.1.1, despite its use of negation, is also easily represented in CLP(R). In Section 5.1.1, we gave an iterative method for finding the result of this condition. In this method, we depended on being able to derive cases where permission was not granted by finding cases where we could not prove permission. This method matches the definition for negation in CLP(R). Thus, the protection condition translates into the CLP(R) clauses:

```
inconsistent([P,Y,T]) :-
        ref_access(P,Y,T), perm_access(S,P,Y,T),
        failed(S1,P1,Y1,T1).
inconsistent([P,Y,T]) :-
        ref_access(P,Y,T), perm_access(S,P,Y,T),
        isinD2([D,S1]),not(isinD2([D,P])),
        perm_access(D,P,Y,T),
        failed(S1,P1,Y1,T1).
inconsistent([X2,X,Y]) :-
        ref_access(X2,Yinstan,readWrite),
        instan(X,Y,Yinstan), decl_access(Y,readOnly).
```

The first two clauses identify those references where no permission at all was given. The first checks permissions by process instances, the second checks permissions by enclosing domains that do not also contain the process instance making the reference. The third clause checks the protection condition on the object itself, and finds any inconsistency when inadequate permission is granted. The `failed` relation is generated during evaluation and saves a derivation of `not(perm_access)`. See Appendix C for the definition of `failed`.

Representing capacity uses CLP(R) constraints and equation evaluation. The formulas for average and maximum load, shown in Section 5.1.2, are translated into deduction rules. These rules operate on lists representing the query rates of clients. The lists are built up during evaluation. The rules derive the aggregate query rates from the lists. The constraints represent the requirements of the querying process. The capacity condition is translates as

```
inconsistent([LH,Result,Y,R]) :-
        retractall(unq(Xfoo)),
        all([F,X], ref_eq(X,F), List),
        breaklist(List, L1, L2), !,
        doeval([R,Result,Y,LH],L1,L2).
```

The first two lines make a list of all references in the specification. The third line breaks out the first server in the list. The final line evaluates the list for the aggregate. The details of `doeval` are shown in Appendix C.

The configuration conditions are translated using the techniques described above. In Section 5.1.3, we described a logical condition for inconsistent configurations and an iterative method for finding inconsistent simple conditions (those that are not defined in terms of a count). This method, like the method for protection conditions, requires only the weak form of negation present in CLP(R). Therefore, we can translate this requirement into CLP(R). For conditions on counts, we use an iterative method based on lists, similar to the method for determining capacity conditions. This method does not use negation, only constraints. So, it is possible to represent the method in CLP(R). The specific translation is similar to the protection and capacity conditions. For details, see Appendix C.

Given these three parts, we have shown that the entire model can be expressed in CLP(R). We have implemented a centralized consistency checker using this approach.

5.3. Incremental Evaluation

So far we have assumed that we will evaluate the specifications in a single, central location. This centralized method has three limitations. First, centralized evaluation requires each administrative domain to provide detailed information about their internal computing environment. Second, it requires a full consistency check each time any single domain's specification changes. Third, the time to evaluate the logic increases non-linearly with the number of logical relations. These problems occur at the domain boundary and share a common solution, incremental evaluation. Incremental evaluation breaks up the consistency proof. The goal of using incremental evaluation is to perform the proof in small increments, and to only cross domain boundaries with the minimal amount of information necessary to complete the proof. We produce a small summary at each increment and perform another proof on this summary and the next part of the specification, repeating the process until all parts have been checked. The effect on incremental changes are discussed at the end of this section.

This section describes a method for performing a NMSL consistency proof incrementally, including logical justification that the incremental proof will prove the same result as the original, centralized proof. We show that in the incremental method, any evaluation we perform will give us a logically complete answer. Because our proof is implemented using CLP(R), we appeal to the representation of NMSL specifications in CLP(R) in our proof. The relative efficiency and performance of the incremental vs. the centralized proofs are also discussed.

5.3.1. Proof of the Incremental Method

The consistency output of the NMSL Compiler is a list of CLP(R) clauses. These clauses are combined with additional evaluation clauses, shown in Appendix C, and evaluated. In the centralized method, the set of all specifications are combined, translated, and evaluated together. In the incremental method, we take the specification for each domain, translate it and evaluate it separately. Each description remains intact when the proof is divided along domain boundaries. In either the centralized or the incremental method, the input to the evaluator is a set of CLP(R) clauses, output by the Compiler.

A specification is *internally* consistent if all of the relationships within the specification of a single domain are met. Specifications are *externally* consistent if the interactions between domains are consistent. We divide our proof into subproofs of each of these conditions. Given both conditions, the incremental method proves the same result as the centralized method.

We show that the proof of internal consistency is identical for both the centralized and incremental method. Since the output of the Compiler for a single specification is identical for either the centralized or the incremental method, the incremental input to CLP(R) is strictly a subset of the centralized input. Since all of the information about internal relationships are present, by definition, all of the internal relationships will be checked in either the centralized or incremental method. Therefore, if the internal relationships were consistent in the centralized check, they will still be consistent in the incremental check. If an internal relationship was not met in the centralized check, the invalid protection, capacity, or configuration condition will still be invalid, because the set of clauses representing the internal interactions of the domain is identical. Therefore, any internal inconsistencies found in the centralized check will be found in the incremental check.

Next, we show that the incremental method evaluates external consistency in the same way as the centralized method. To determine the same results for external inconsistencies requires that we identify those clauses generated by the Compiler that correspond to external relationships. Each of the clauses shown in Section 5.2 are *supported* by other clauses. Support is defined in two parts. First, when proving the an instance of the head of the clause, the truth of instances of the relationships in the tail must first be proved, so clauses that prove those instances support the head. Second, if the clause is used in conjunction with other clauses in a proof, all of those clauses support each other. An example of the second type of support is the set of `ref_eq` clauses relating the same client and server; such clauses support each other. Below we show a method for identifying exactly those clauses denoting external relationships and their supporting clauses. We call this set of external relationships a domain's `external summary`. We can apply this method to the clauses representing each domain's specification. We then combine the external summaries. Assuming this method for find exactly those clauses relating to external relationships, we can prove external consistency. Since the algorithm will generate all of the external relationships, the incremental method will prove the same result as the centralized method. If the centralized method proves consistency, the incremental method will also prove consistency, because all of the clauses representing external interactions will be present in the external summary. Likewise, if the centralized method proved external inconsistency, the exactly those clauses that were inconsistent in the centralized method will be present in the external summary, so the incremental method will prove inconsistency.

5.3.2. A Domain-Based Method for Finding External Summaries

The method for finding external summaries depends on knowledge of the Compiler's consistency output. The method inspects each CLP(R) clause output by the compiler. If the clause is part of the external summary, the method finds the clause's supporting clauses, and adds all of these clauses to the external summary. We address each clause output by the Compiler, as discussed in Section 5.2, in turn. Then, we show an algorithm to find an external summary.

In this method, we make two assumptions about the clauses used in consistency evaluation. First, all clauses denoting objects (the MIB) are included in each domain's specification. This is reasonable because no interaction can take place without a common set of objects. Second, the evaluation rules, listed in Appendix C, are replicated, with one copy per domain. This is also reasonable, because we cannot break up the consistency proof unless all domains use the same evaluation rules.

Consider clauses denoting objects.

```
contains([[1,3,6,1,2,1,1],[1,3,6,1,2,1,1,1]]).
decl_access([1,3,6,1,2,1,1,1],readOnly).
```

Such clauses do not refer, in themselves, to external relationships. Therefore, these clauses are not candidates for the external summary. Furthermore, because we assume that all specifications define the same set of objects, we can avoid ever making such clauses part of the external summary.

Next, consider clauses that export protection or capacity.

```
perm_access(S2,'wisc-cs',Y,readOnly) :-
        instanD(S1,process('snmpdReadOnly'),S2),
        instan(S2,[1,3,6,1,2,1 | V1],Y).
perm_eq('wisc-cs',Y,1,10) :-
        instanD(S1,process('snmpdReadOnly'),S2),
        instan(S2,[1,3,6,1,2,1 | V1],Y).
instan(P1,[1,3,6,1,2,1],[P1,1,3,6,1,2,1]) :-
        instanD(S2,process('snmpdReadOnly'),P1).
```

For the `perm_access` clause, we can inspect the second parameter of the relationship, the destination of the protection export, and determine if that destination is internal or external. If the destination is external, the clause is added to the external summary. We must locate the clauses that defines the instantiation of the process, `S2` by the `instanD` clause, and the object instance, `Y`. The clause defining the object instance is the `instan` clause shown at then bottom of the CLP(R) code fragment. The `instan` clause depends on the same `instanD` clause as the `perm_access` clause. We can inspect all of the `instanD` clauses output by the compiler, and pick those that instantiate the process named in the `perm_access` clause. The `instanD` clauses have no tail, so they have no supporting clauses. Because `perm_access` clauses are referenced when determining domain-wide access permissions,

```
        inconsistent([P,Y,T]) :-
                ref_access(P,Y,T), perm_access(S,P,Y,T),
                isinD2([D,S1]),not(isinD2([D,P])),
                perm_access(D,P,Y,T),
                failed(S1,P1,Y1,T1).
```

we must include all of the `containsD` clauses that support the `perm_access` clause in this role. Recall that domain containment is implemented using lists, so the `isinD2` relations above (shown in detail Appendix C) indirectly reference `containsD` relations. To find all supporting `containsD` relations we take each `instanD` clause we added to the summary, above, and find the domain that contains the network elements listed in these `instanD` clauses. For each such domain, we recurse and find the containing domains. Each clause listing these containment relationships is added to the summary.

Next, consider the `perm_eq` clause shown in the earlier CLP(R) fragment. The destination of the exported capacity is the first argument. After determining if the destination is external, we follow the same supporting clauses as described for the `perm_access` clause. In addition, if the destination domain of the `perm_eq` clause contains the local domain (directly or transitively), part of the capacity listed can be used by the local domain. To account for this locally used capacity, we must include all local `ref_eq` clauses that reference this `perm_eq` clause in the external summary. Such clauses are identified by inspecting the process referenced in the `ref_eq` clause. The supporting relationships for the `ref_eq` clauses are detailed below.

Next, consider clauses representing queries.

```
        ref_access(P2,Y1,readOnly) :-
                instanD(S2,process('snmpxmon'),P2),
                instanD('gw',process('snmpdReadOnly'),P1),
                (instan(P1,[1,3,6,1,2,1,2,2,1,8],Y1)).
        ref_eq(P2,Y1,0.02,1,avg) :-
                instanD(S2,process('snmpxmon'),P2),
                instanD('gw',process('snmpdReadOnly'),P1),
                (instan(P1,[1,3,6,1,2,1,2,2,1,8],Y1)).
        ref_eq(P2,Y1,0.98,0,avg) :-
                instanD(S2,process('snmpxmon'),P2),
                instanD('gw',process('snmpdReadOnly'),P1),
                (instan(P1,[1,3,6,1,2,1,2,2,1,8],Y1)).
        queries(P1,P2) :-
                instanD(S1,process('snmpxmon'),P1),
                instanD('gw',process('snmpdReadOnly'),P2).
```

If a `ref_access` clause is external, then the network element shown in the second `instanD` relationship,

```
                instanD('gw',process('snmpdReadOnly'),P1),
```

will be external. Given an external `ref_access` clause, we must find the clauses that instantiate the process listed in the first `instanD` relationship,

```
                  instanD(S2,process('snmpxmon'),P2),
```

and add each instantiation to the external summary. As in the cases shown above, we must also add the corresponding `instan` clauses to the external summary. Recall that we can find all relevant clauses by inspecting the `instanD` relationship used in the body of the `instan` clause (see above). Instances of the second `instanD` clause are not added to the external summary, because it denotes an external instantiation. The external instantiation is part of another domain's external summary.

The `ref_eq` clauses are handled similarly. Because `ref_eq` clauses depend on exactly the same clauses as do `ref_access` clauses, we need not find the supporting clauses for `ref_eq` clauses, because `ref_eq` clauses are always are present in conjunction with a `ref_access` clause. Note, however, that we must add the clauses representing all of the modes of the query distribution to the summary. The Compiler generates these in order, so finding all relevant `ref_eq` clauses is easy.

The `queries` clause is treated just like the `ref_access` clause, and the appropriate instantiations, as shown by the first `instanD` relationship in the `queries` clause are also added to the external summary. The queries clause is used only by `provides` and `requires` clauses denoting peer configuration requirements. Therefore, we must find clauses such as

```
     provides(Y2, 'protocol', '=', 'snmp', X) :-
             instanD(S1,process('snmpdReadOnly'),Y2),
             queries(X,Y2).
```

that refer to the same process in the `instanD` clause as the process listed in the `queries` clause. Such clauses support the queries clause, and are added to the external summary. The relationships used in the body of such clauses have already been added to the external summary.

Next, consider configuration conditions.

```
     provides(Y2, 'protocol', '=', 'snmp', X) :-
             instanD(S1,process('snmpdReadOnly'),Y2),
             queries(X,Y2).
     requires(Y2, 'root, '=', 'true', X) :-
             containsD(['S1,'wisc-cs']),
             Y2 = S1.
```

There are two ways such clauses can be part of the external summary: if the clause denotes a peer configuration condition and the peer is external, or if the clause denotes an enclosing configuration condition on an external domain. The first case was handled above, when looking for clauses that support a `queries` clause. The second case can be determined by inspection. If the destination domain listed in the `containsD` relationship is the parent of the domain being summarized, then the clause is added to the external summary. The `containsD` relationship supports the configuration condition, so it must also be added to the summary.

Finally, consider domain containment.

```
     containsD(['wisc-cs','romano']).
```

Any containment clauses that are needed in the external summary will already have been included in support of `perm_access`, `providesor requires` clauses. Therefore, the remaining `containsD` clauses need not be included.

In two cases, we can further summarize the external capacity distributions. First, if there are `ref_eq` clauses for several local clients referencing the same external server, we can aggregate the client's rate distribution, as discussed in Section 5.1.2, and include this aggregate distribution in the summary, rather than the distribution of the individual clients. Recall that aggregation can introduce one-sided error in the aggregate distribution. However, because this aggregation is performed anyway, during the course of the proof, we can aggregate locally without adding additional error.

Second, if a `perm_eq` clause exports permission to a containing domain, and there are local references to this `perm_eq` clause, we can summarize the locally used capacity in the same way, and include the aggregate distribution of the locally used capacity in the summary.

5.3.3. An Externalization Algorithm

This discussion leads to the following algorithm for identifying the clauses from a NMSL specification that are involved in cross-domain interactions. The output of this algorithm, Algorithm 5.1, is the external summary of a single domain. This algorithm assumes that the database of clauses includes only the clauses for a single administrative domain. Algorithm 5.1 first builds a tree of containment and instantiations (line 5). Clauses that are entered into the tree are also marked as being local or remote, depending on their containment (anything contained in the local domain is local, everything else is remote). The algorithm processes the clauses a second time, looking for those involved in remote interactions. Local and remote interactions are determined (lines 8, 13 and 22) using the domain tree built in the first step. For each remote interaction, the algorithm determines the type of interaction (protection, capacity or configuration), and searches the clause database for all supporting clauses (line 10), using the requirements detailed above. Recall that supporting clauses describe instantiations, domain containment, peer configuration, and query distribution. For exported capacities, the amount of capacity used by local queries is looked up in the clause database, aggregated, and included in the external summary (lines 16 through 18). Each time a remote reference is located (line 22) the distribution mode is combined with other modes already in the summary, to form an aggregate distribution within the summary. The output is the set of clauses for the external summary (line 25).

Efficiency is an important issue in implementing this algorithm. In the first phase, we iterate over the entire database of N clauses. Entering a clause into the domain tree takes time to find the right place in the tree to add the clause. The time depends on the depth of the tree. The depth of the tree is bounded by the height of the domain containment hierarchy, which is constant over the course of a proof. Thus, the total preprocessing time is $O(N)$.

Given: A set of NMSL clauses for a single domain and external interactions

Compute: The external summary

Algorithm:

```
BuildSummary:
    DomInstTree = ∅; Summary = ∅;
    for each NMSL clause f
        if ( type of f is "ContainsD" or "InstanD" )
            insert f in DomInstTree;
    for each NMSL clause f {
        if ((type of f is "perm" or "ref" or "provides" or "requires" or "count")
                    and ( dest of f is not local) ) {
            append f to Summary;
            for each clause e that supports f
                append e to Summary;
        }
        if ( (type of f is "perm_eq") and ( dest of f is not local) ) {
            if ( dest domain of f contains local domain ) {
                Aggregate = ∅;
                for each clause, e, that references f
                    combine e with Aggregate;
                add Aggregate to Summary;
            }
            append f to Summary;
        }
        if ( ( type of f is "ref_eq" ) and (dest of f is not local) )
            combine f with Summary;
    }
    output Summary;
```

Algorithm 5.1. Build an external summary

In the second phase, we iterate over the database of N clauses again. We assume that the database is accessed with a hash table, with keys based on the type of the clause, the source domain or process specified by the clause, and the clause's destination, an object or another process or domain, as needed for finding supporting clauses, discussed above. Determining if a clause is a local or remote interaction requires a constant-time lookup in the clause database and a check to see if the domain of the clause in question was marked local or remote. For each clause denoting a remote interaction, we must find all supporting clauses. Looking up each supporting clause is constant time, because we hash on the parts of the clause used for determining support. Furthermore, if we remove a supporting clause from the fact database when we add it to the external summary, we never process such clauses twice. This means that over the entire list of N clauses, the time to process each clause is constant. Therefore, the complexity of finding these conditions is $O(N)$.

For each clause denoting the export of a capacity, we must determine if part of the capacity is taken up locally (line 14), which is a constant-time lookup in the clause database. For such clauses, we must determine the aggregate load of all local accesses (lines 16 through 18) and add this to the external summary. The time to do this requires looking up the clause for each mode of the rates (constant time), adding the mode into the aggregate (constant time). There will be one clause for each instance of each mode of the rate in the clause database, so the bound on the number of clauses we must look up is N. The maximum number of modes per instance is m, which is a bounded by a constant, as shown in Section 5.1.2. The time to incorporate a new mode into the aggregate is constant time, also shown in Section 5.1.2. If we remove these supporting clauses from the clause database, as we did above, we never compute an aggregate more than once. This reduces the time to process each capacity export to constant time, over the entire fact list. Therefore, the time to calculate all capacity conditions is $O(N)$.

Given that the bound for computing protection and configuration conditions is $O(N)$, and the bound for computing capacity conditions is also $O(N)$, we see that the complexity of calculating all remote interactions is $O(N)$.

Since we can produce external summaries, we can break up the consistency proof along domain boundaries. Furthermore, we can perform the hierarchical proof we outlined at the beginning of this chapter. Each bottom level domain's specification can be checked for consistency separately and in parallel. Their external summaries can be combined at the next higher level in the domain tree and checked in a later step. This process is repeated up the tree until the root is reached. If there is to be a savings in elapsed time for the incremental consistency check, the size of the summary for each domain must be much smaller than the size of the full specification for that domain. If the size of the summary is close to the size of the full specification, the elapsed time to check consistency approaches the time for the centralized method. We show in Chapter 7 that the specifications in our test implementation have the property that their external summaries are small. Additional details of the hierarchical implementation are also given in Chapter 7.

5.3.4. Incremental Re-evaluation

Once a consistency proof is completed, administrators can change their specifications. After any changes, consistency must be re-evaluated. The incremental method allows us to reduce the amount of work needed for re-evaluation. To reduce the workload, we must save the external summary for each domain's specification.

When a specification is changed, a new external summary must be determined. If this external summary is identical to the old external summary, then only the local, internal proof must be performed. We proved external consistency in the previous consistency check, and since no external conditions changed, the specification is still externally consistent. If the external summary changes, we must re-evaluate external consistency. When using the hierarchical method, we can still achieve a savings in this case, because we propagate the external summary up one level in the domain hierarchy. At this higher level, we determine the external summary. If it is unchanged, we can stop the proof after evaluating consistency at this level. In the worst case, we must keep re-evaluating until the root is reached. However, this is no worse than not taking advantage of the knowledge included in the external summaries.

5.4. Summary

In this chapter, we have presented the model for consistency of NMSL specifications. This model is based on three conditions for consistency, protection, capacity and configuration. We described methods for evaluating specifications using this model is a centralized manner. However, centralized evaluation is limited, because centralized evaluation can severely limit domain autonomy. To deal with this limitation, we introduced an incremental method for evaluating consistency, based on the hierarchical organization of domains. We showed that this incremental consistency check can solve the same problems as the centralized check.

The bottom-up hierarchical consistency check is not the only type of incremental method possible. One alternative is to pass inter-domain relationships to the peer domain, rather than the parent domain. Restrictions from the parent could be passed down the hierarchy, rather than up. This method might be useful if each parent has many children, or if all domains are at the same level (that is, there is no hierarchy). Using the hierarchical method described above, all inter-domain relationships would be evaluated at a central site. The peer-domain method can also be helpful if the parent domain is not trusted with information about peer interaction. Because of these advantages, we plan to investigate this method in the future.

Chapter 6

Prescriptive Aspects

Once a specification is determined to be consistent, the specification can be used to configure the network management processes and the newly installed configuration can be monitored and enforced. A NMSL Configuration Installer takes output from the NMSL compiler and uses the output to configure a network manager. The NMSL compiler does not configure a network manager directly, because configuration requires knowledge of data formats, protocols and authentication issues. A Configuration Installer is, therefore, a separate module that interprets the configuration output of the compiler and performs the implementation-specific actions necessary to install the configuration in a network management process. Enforcement means detecting and (possibly) taking action against processes in the system that do not behave according to their specification.

The operation of the prescriptive mode is as follows. After a specification has been shown to be consistent, the NMSL compiler is re-run with a parameter requesting configuration output of a specific type. A Configuration Installer takes the configuration output and transmits it to each appropriate network element or process. Configuration is achieved, in most cases, by initiating a connection to a network management process (specified in the NMSL specification) on each affected network element, authenticating the Configuration Installer as a trusted process, and sending, via the normal network management protocol, the configuration information. Objects representing the configuration information were added to the MIB. In other cases, the data might be copied, in the form of a file, to the affected network element, or even sent via electronic mail to the administrator of that network element.

Once the configuration is installed in the system, it can be enforced. Enforcement entails monitoring the running system to determine if the processes are operating within the specified bounds for protection, capacity and configuration. If not, the system administrator can be notified, and, in some cases, corrective action can automatically be taken. We now address both areas of installation and enforcement. Enforcement is not within the scope of this thesis, so the enforcement details are all areas of future work.

6.1. Installing Configurations

We have investigated two issues in installing configurations. First, how one installs configurations in a running system efficiently, without infringing on the autonomy of administrative domains. Second, how one should implement the installation. The implementation depends on a solution to the issue of autonomy.

Installing configurations must be performed in a distributed way. Centralized configuration would require a single administrative domain to have control over all other administrative domains. Given this control, the single Configuration Installer would have to update the configuration of all parts of the network management system. Configuring from a central site is not viable for two reasons. First, it violates the autonomy of the administrative domains. Second, it is likely that the overall management specifications will be updated frequently, on the order of minutes for a large group of domains. With such a high frequency, it would be common for an installation to still be running when the next installation begins. However, a specification for an individual administrative domain will not be updated as frequently, on the order of days or weeks. If we distribute installation, we can preserve the autonomy of the administrative domains, reduce the frequency at which individual parts of the management system get reconfigured, and decrease the elapsed time it takes to install a configuration. In Chapter 5, we introduced the idea of an incremental consistency check, allowing us to implement a consistency checker distributed over the domain hierarchy. We can distribute the automated configuration over this same hierarchy.

Recall that we required administrators to give enough information in their specifications so that a local consistency check, using the centralized method, can be performed. A side effect of this requirement is that enough information is present to perform local configuration installation. However, to avoid installing inconsistent specifications, the local domain must wait until the parent domain has passed back the success of the consistency check before beginning installation. This method allows the most autonomy possible, because it does not require domains to give the ability to configure their processes to other domains. In our implementation, we follow these steps (check for local consistency, wait for remote results, install locally).

There are several choices for installing the configuration into a running system. All of the methods assume that the Configuration Installer can authenticate itself with the systems and processes that it is configuring. One method is to use a standard network management protocol, such as SNMP or CMIP, to install the configuration in an extended MIB. This method is preferable because the processes of the network management system already speak a management protocol, so no additional management interfaces are needed in the rest of the system. In our experience, using a management protocol is also the fastest way of installing a new configuration. Another method would be to use a remote execution service, for example, the Berkeley UNIX remote shell (rsh) service. Remote execution services generally have higher overhead management protocol, and are not always available. For example, a dedicated router may only allow configuration via the management protocols it supports. Furthermore, because interactions are not directly with the management process, it is not as easy to determine if the installation was successful. The final method, using electronic mail or a similar service, is also the least reliable. With electronic mail, the Configuration Installer has no control over when or if the new configuration will be installed. Such installations would probably be performed by a human system administrator, and humans occasionally act even less predictably than software. In all of these approaches, there is the possibility that the installation will fail, however, the ability of the Installer to compensate for failures decreases as we move to higher and higher level protocols. It is also reasonable for the Installer to support more then one method for installing configurations, so if one fails, it can try another.

The NMSL Configuration Installer takes an optimistic approach to installing configurations, no matter which method is used for installation. The approach is optimistic in that it allows the installation to go ahead, assuming the configuration of the network will be consistent by the time the installation is complete. The installer assumes the installation will succeed, and if the installation fails, the installer notifies the administrator of the failure. This approach allows as many of the processes and network elements as possible to be configured correctly. Also notice that the network management system is technically in an inconsistent state while a new configuration is being installed. In the middle of the installation, some parts have the new configuration, some have the old configuration. As soon as all parts of the system are brought up to date, the configuration will be consistent again. The behavior of management interactions taking place during reconfiguration is undefined. The implementation should make it possible to detect this being-reconfigured state, so any erroneous behavior can be ignored and interactions can be retried when reconfiguration completes.

6.2. Enforcing Specifications

Enforcing NMSL specifications means making sure all processes in a running network management system operate within the bounds of the specification (operation is defined in terms of the process's interactions). Enforcement requires the ability to monitor and modify the state of a running management system. If that ability is not available, it is possible for the system to be inconsistent, despite the consistency of the specification.

An issue of enforcement is determining how often and in what way the behavior should be monitored. If one monitors too often, it affects the performance of the system. If one does not monitor frequently enough, the system could remain inconsistent for a long time for the problem is noticed. To monitor the network management system, we can either make queries of the processes of the management system itself or, for some statistics and networks, we can passively monitor the traffic on the network. Passive monitoring reduces the effect of the Enforcer on the network's performance, but cannot be used for monitoring the activity within a network element itself. For example, we cannot tell if a management process is overloaded because it's specified capacity was too high just by monitoring the requests sent to the management process over the network. In this case, the number of requests are within the specified limits, but the specification for the recipient was incorrect. These are not new issues; they are present in any network management decision.

In the network management systems with which we have experience, only some processes can be monitored. All network management agents can be monitored, but managers often do not include a monitoring capability. In such systems, we can only enforce the specification on the agent processes. We can monitor the query rates, access types, and configuration requirements, but, since we only have one-sided monitoring, this does not completely solve the problem of enforcing the specification on the processes initiating the queries. For example, if too many processes are sending queries, this may not be noted in the agent's statistics, because the overload of queries might get dropped or lost in the network itself. Even passive monitoring of the network can miss such overloads. In the future, all network management processes should define a standard interface to make their status available via the management protocols. This would allow us to enforce the specification on all processes.

6.3. Summary

Distributing the configuration of the network management system is as important as distributing the consistency check. Distribution allows the NMSL system to provide an automated configuration installer, but does not require administrators to give up complete control of their domains. We have shown that breaking up configuration along domain boundaries achieves this goal. Full enforcement of a NMSL specification in a running system requires access to the current state of all processes in the system, for both monitoring and control. If either of these is not present, the system can only give limited guarantees on the consistency of the running system.

Chapter 7

Implementation and Performance Measurement

In this chapter, we describe the implementation of the NMSL system and provide performance measurements for the system. There are four parts to the system: compilation, incremental consistency checking, configuration installation, and configuration enforcement. We present the implementation and performance of the NMSL Compiler and Consistency Checker. For the compiler, we present measurements of its speed. For the Consistency Checker, we present both centralized and incremental performance results. We first describe the time to check several domain specifications in a centralized manner. We then describe the times to separately evaluate and summarize the domain specifications, followed by a hierarchical check of the parent specification with the summaries generated in the first step. We compare the performance of the centralized and incremental methods. Memory utilization and the size of the summaries are also presented. We also outline our implementation of the Configuration Installer and discuss how one might implement a Configuration Enforcer.

7.1. The NMSL Compiler

The NMSL Compiler consists of three phases: preprocessing, parsing, and semantic analysis. The input to the first phase is a textual specification. Preprocessing is performed using the C preprocessor[21]. The second phase reads the preprocessed input, checks its syntax and reorders the descriptions in a way that simplifies semantic checks. This reordering causes network element and domain descriptions to be checked after data module and process descriptions. In the third phase, the specification is checked for semantic errors, and compiled into either CLP(R) clauses for the Consistency Checker or configuration instructions for the Configuration Installer. The type of output depends on the options given to the NMSL Compiler. Some inter-description checks are also made, most importantly, checks of process instantiations within network element descriptions.

The semantic checks and output generators are coded in an extension language. The Compiler reads in the semantic and output generating routines, written in the extension language, when it is initialized. This extension language is a simple, interpreted language tailored to the syntax of NMSL. An example of the extension language and a more detailed description if its syntax is given in Appendix D. The extension language is a list of procedures. Each procedure is introduced by a predicate and has two or more sub-procedures. One sub-procedure performs semantic checks; the others generate output of different types. The parse tree for each clause and declaration is tested against each predicate in the extension file. If a match is found, the semantic processing sub-procedure is executed. Then the appropriate output sub-procedure is executed, depending on the compile-time options. There are two benefits to using an extension language. First, the extension language reduces the time spent in developing the compiler. Second, it allows an easy way to incorporate new output types into the compiler.

7.1.1. Consistency Output

One type of compiler output is the collection of CLP(R) clauses for the Consistency Checker. Each clause in NMSL is translated into one or more CLP(R) clauses. Each clause states the information present in the clause and relates the information to the description containing the clause. Recall that descriptions group clauses and that there are four description types: type descriptions, process descriptions, network element descriptions, and domain descriptions.

In Figure 7.1, we show an example of translating part of a NMSL specification into CLP(R). This example shows the type of output the compiler generates for consistency checking. The specification fragment is taken

```
process "snmpPing" ( Proc, SysName ) ::=
  queries Proc {
    on SysName;
    requests mgmt.mib.system.sysUpTime;

    rate avg {
      mode 1 0.02 1 qps;
      mode 2 0.98 0 qps;
    }

  }
end process "snmpPing".
```

$\downarrow$

```
ref_eq(P2,Y1,0.02,1) :-                          (1)
  instanD(S2,process('snmpPing'),P2),            (2)
  instanvar(S2,[P2,'SysName'],Shost),            (3)
  instanvar(S2,[P2,'Proc'],Svar),                (4)
  instanD(Shost,process(Svar),P1),               (5)
  instan(Svar,[1,3,6,1,2,1,1,3],Y1).             (6)
ref_eq(P2,Y1,0.98,0) :-
  instanD(S2,process('snmpPing'),P2),
  instanvar(S2,[P2,'SysName'],Shost),
  instanvar(S2,[P2,'Proc'],Svar),
  instanD(Shost,process(Svar),P1),
  instan(P1,[1,3,6,1,2,1,1,3],Y1).
```

Figure 7.1.
Translation from NMSL into CLP(R)
A rate clause expanded into CLP(R)

from the process description in Figure 4.2. In the example, the `rate` clause shown in the upper box is translated into the CLP(R) statements shown in the lower box. The other statements are shown to give the reader some context.

The two modes of the rate clause translate into two CLP(R) statements. We explain the details of this translation for the first mode; the second mode is translated analogously. Line 1 states that some process instance `P2` will reference some object instance `Y1` 2% of the time with one query per second. Lines 2 through 6 relate the clause to the containing process description. Since the rate is being listed for a query, these lines also relate the clause to the destination of the query. Line 2 says that `P2` is any instance of the `snmpPing` process on any network element, `S2`. `S2` will be constrained by clauses generated for `process` clauses found in network element descriptions. Lines 3 and 4 constrain the network element and process identifiers, `Sysname` and `Proc` in the upper box, to the constant values, `Shost` and `Svar`, respectively. `Shost` and `Svar` identify the parameters specified when the process is instantiated. Line 5 constrains the process instance `P1` to the instance of the process identified by `Proc` on the network element identified by `Sysname`. Line 6 constrains `Y1` to the object instance of `mgmt.mib.system.sysUpTime` found in process `P1`.

7.1.2. Configuration Output

The other type of compiler output is a collection of configuration instructions. We have currently implemented Compiler output procedures that generate output to configure ISODE SNMP agent processes[31, 32]. The output is passed to a Configuration Installer to reconfigure the management system. The output is textual, and consists of configuration files for the agents. These configuration files include such data as export types and interface definitions. They also include configuration conditions such as the name of the person to contact about the agent process. For export types and interface definitions, we output the information described in `exports`, `queries`, and `interface` clauses. Configuration conditions are specified using `provides` clauses.

An example of configuration output is shown in Figure 7.2. This output is meant to configure an ISODE SNMP agent process, and the specification fragment is taken from the network element description in Figure 4.2. The `exports` clause translates into the first and second lines of output. Only the domain name (`"wisc-cs"`) and the permitted access type (`ReadOnly`) are used by this configuration file format. The `queries` clause translates into the third line of output. The process and network element names are used to configure the recipient of trap messages. The notation of the output is the ISODE configuration file format. The token `0.0.0.0` represents an access code. `0.0.0.0` means no access code is required. The token `1.17.3` is used to link `community` and `view` statements. The `community` statement defines a community name and its permissions. The `view` statement lists the objects that make up the view. The `trap` statement lists the process to which traps should be sent, and the network element instantiating the process.

```
process "snmpdReadOnly" ::=
  exports to "wisc-cs" {
    objects mgmt.mib;
    access ReadOnly;
    rate max {

    }
  }
  queries "snmpt" {
    on "romano";
    traps any;
  }
end process "snmpdReadOnly".
```

$$\downarrow$$

```
community "wisc-cs" 0.0.0.0 readWrite 1.17.3
view 1.17.3 mib-2
trap snmpt romano
```

Figure 7.2.
Translation from NMSL into Configuration Output
An exports and a queries clause translated into the ISODE SNMP agent configuration file format

7.1.3. Compiler Performance Measurement and Discussion

We first measured the performance of the NMSL Compiler. To test compiler performance, we compiled several sample specifications. Each specification included a module describing the SNMP MIB, two agent processes (A_1 and A_2), three manager processes (M_1, M_2, and M_3), two domains (one local and one remote), and N network elements. In our measurements, we varied N from 2 to 20. One network element in each specification is in the remote domain. This network element instantiates processes A_1 and M_1. Each network element in the local domain (the remaining $N-1$) instantiates the other agent process, A_2. One network element in the local domain instantiates the other two manager processes, M_2 and M_3. This style of specification is meant to be representative of a specification for a real network management system. In later sections, we refer to this data set as the Compiler Test data set. Figure 7.3 illustrates a specification with 6 network elements. These specifications are modeled after the NMSLTEST specification, shown in Appendix B.

The size of the specifications vary from 99 lines (for 2 network elements) to 473 lines (for 20 network elements). Each additional network element adds about 30 lines of text. Each line of text corresponds roughly to a NMSL clause.

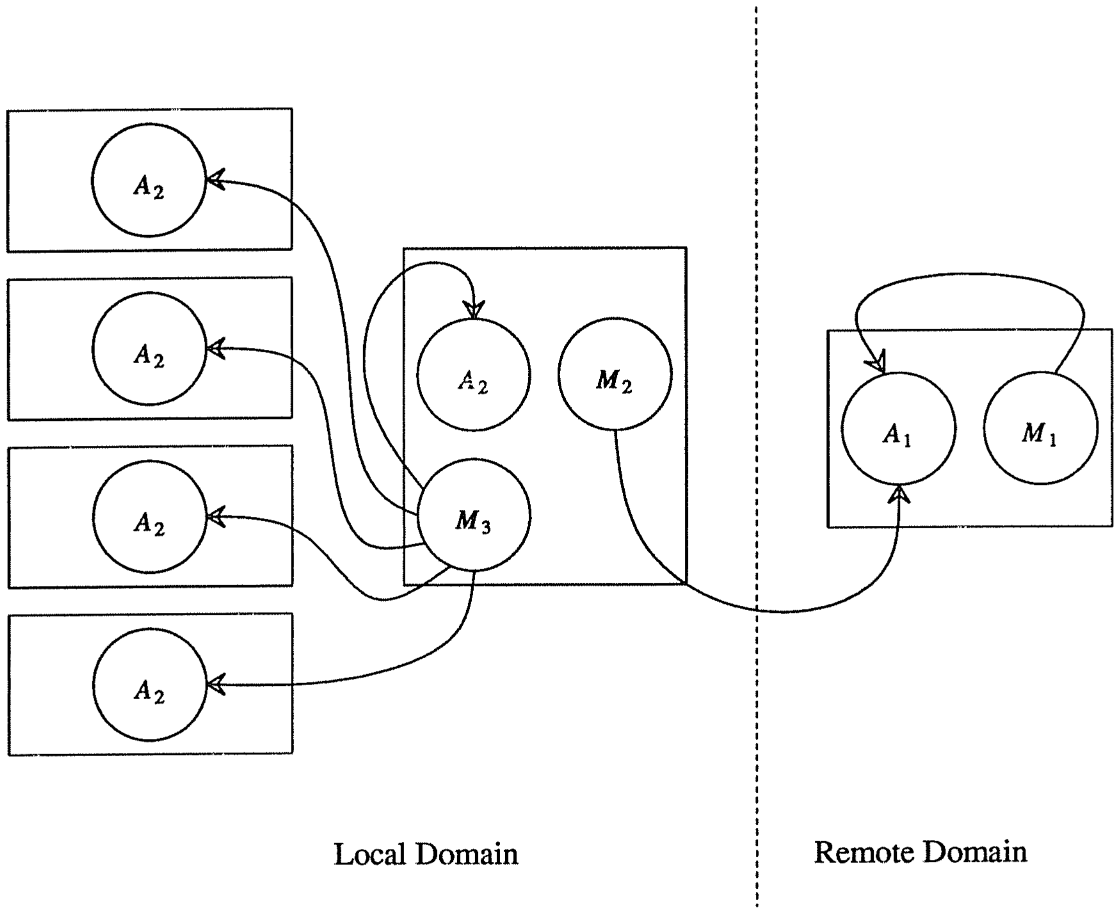

Figure 7.3.
The Components of a Six Network Element Specification
Boxes are network elements, circles are processes.
Arcs show Interactions between Managers and Agents.

Figure 7.4 shows the speed of the compiler. The data point for zero network elements is the time to compile an empty specification. This graph shows that the time to compile a specification varies linearly with its size measured in network elements. The size of the output of these tests, measured in CLP(R) clauses, also grows linearly with the number of network elements in the specification.

These measurements show that the compiler performs reasonably well for the test input. Although these measurements test small specifications, we feel it is reasonable to expect the time to remain proportional to the size of the specification as the size increases. The size of systems we hope to eventually specify with NMSL is on the order of thousands or hundreds of thousands of network elements. The data shown in the graph implies an average of 0.26 seconds per network element. For a 100,000 network element specification, then, we would

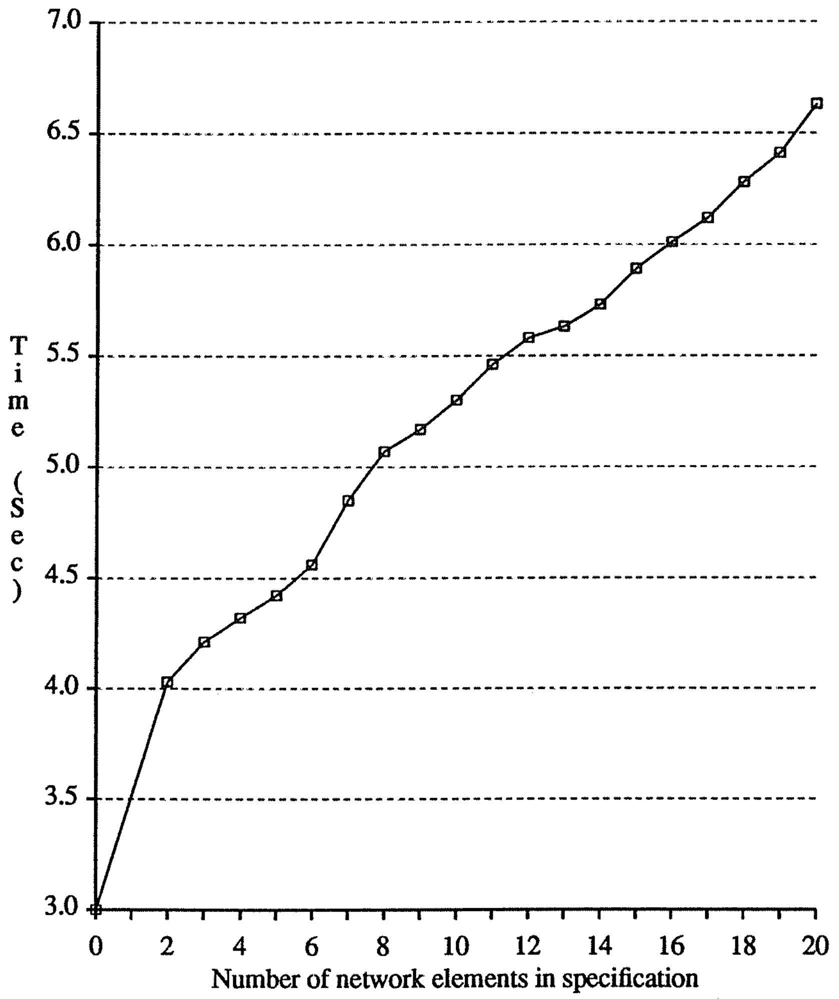

Figure 7.4.
Compile times for NMSL Specifications
Varying the number of network elements in the Compiler Test Specification
Tests run on a DECstation 3100 with 20 Mbytes of memory

expect the compilation time to be over 7 hours. However, such a large network management system would be made up of several, possibly hundreds of domains. The specification for each domain is written separately. Therefore, we compile the specification for each domain separately and use the incremental method for checking the set of specifications for consistency. Separate compilation preserves the autonomy of the domains and allows parallel compilation.

7.2. The Consistency Checker

We implemented an incremental Consistency Checker, as described in Chapter 5. There are two parts to this implementation, a Consistency Evaluator and an Externalizer. The Evaluator determines the consistency of a specification that has been translated into CLP(R). The Externalizer determines the external summary of a specification for a single administrative domain.

These two tools are used to implement the hierarchical proof we described in Chapter 5. First, a domain administrator writes a specification for the local domain and its interactions with other domains. The administrator then uses the NMSL Compiler to translate this specification into CLP(R) clauses. The Consistency Evaluator takes the CLP(R) clauses as input, adds CLP(R) evaluation rules to this list, and uses the CLP(R) interpreter to find inconsistencies. If the specification is consistent, the Externalizer is executed. The Externalizer's input is the same as the input to the Consistency Evaluator, the CLP(R) rules for a single administrative domain. The Externalizer uses Algorithm 5.2 to compute the external summary of an administrative domain. Its output is the local domain's external summary in the form of CLP(R) clauses. This output is sent to the administrator of the parent domain. The administrator of the parent domain combines the external summaries for all of the child domains with the CLP(R) clauses for the specification of the parent domain and then evaluates this combination of clauses using the Consistency Evaluator. This process continues until the root of the tree is reached. The consistency or inconsistency results are propagated back down the tree. Most of these steps can be automated, but all are executed manually in our current implementation.

7.2.1. Performance of Consistency Checking

This section describes the performance results for the Consistency Checker. We present measurements of the execution time and memory usage of the CLP(R)-based Evaluator. We also present measurements of the size of the Externalizer's output and the speed of the Externalizer. We compare the performance of using the Evaluator in a centralized way with the incremental method.

We use four sample specifications as test data to perform these measurements. The first specification is the output of the Compiler Test discussed in Section 7.1.3. The other three specifications are written for real network management systems. One is for a small network management system used in testing the NMSL system. Another is for WISCNET, the Wisconsin educational backbone network. The third is for CICNET, the midwest regional backbone network. All of theses specifications are similar in content. See Appendix B for the full specification of the NMSLTEST system. The WISCNET and CICNET systems are similar to the NMSLTEST system, except the number of network elements are different, and the names for network elements and certain processes are different. Since they are so similar, the text of their specifications is not included.

The NMSL test network management system (NMSLTEST) consists of seven network elements and two domains. There are two agent processes, a trap (alarm) handling process, and a manager process. One agent is instantiated on each of six of the network elements. NMSLTEST also queries an agent within the CICNET domain. The manager process is instantiated on one network element and performs "keep alive" tests on the network elements it manages. It also performs "keep alive" tests on the gateway to CICNET. The trap handler is instantiated on one of the network elements and accepts traps from any of the agents. The specification is 190 lines long.

WISCNET contains a single domain with 22 network elements. The management system manages these network elements and queries the local CICNET gateway (in the CICNET domain). Each network element instantiates an agent process; one element instantiates a trap handling process and two manager processes. One manager process sends "keep alive" queries to the network elements in the network and of the CICNET gateway, and displays the status of the network elements for the network administrator. The other manager process gathers performance statistics. The trap handler accepts traps from the agent processes. The specification for WISCNET is 683 lines long.

CICNET is comprised of a single domain with 13 network elements. Each network element instantiates an agent process; one network element instantiates a trap handler, two monitoring processes and a performance gathering process. Both of the monitoring processes send "keep alive" queries to each network element in the network. The two monitoring processes use different protocols for their queries. The specification for CICNET is 494 lines long.

We divide our performance measurements into centralized and incremental measurements. We benchmark the Centralized Evaluator using the Compiler Test data set. We then show the time to centrally evaluate the three-domain data set. For this test, we combine the specifications for the three network management systems into a single, large specification. Next, we present the incremental measurements. We show the relative sizes of the input and output to the Externalizer for the three-domain data set and for a specification from the Compiler Test data set. We also show the time to separately evaluate and externalize the specification for each domain in the three-domain data set. Finally, we compare the time to execute the centralized and the incremental evaluators using the three-domain data set.

Figure 7.5 summarizes the measurements of centralized evaluation time, for the Compiler Test output, varying the number of network elements in the specifications from 2 to 20. There are three sets of measurements shown in this graph, one showing the elapsed time on a Sequent Symmetry with 40 Megabytes of main memory[6], one showing the elapsed time on a DECstation 3100 with 20 Megabytes of main memory, and the last showing the combined user and operating-system time on the DECstation 3100. We see that the evaluation time increases faster than linearly, though the degree of the increase cannot be accurately determined because it depends on the implementation of CLP(R). The rapid increase in elapsed time on the DECstation 3100 from 16 to 17 network elements is because of increased paging activity due to the high memory utilization of CLP(R). The combined user and operating system time on the DECstation 3100, which was essentially the same as elapsed time before

[6]We used the Sequent Symmetry for its large main memory, not for its parallelism.

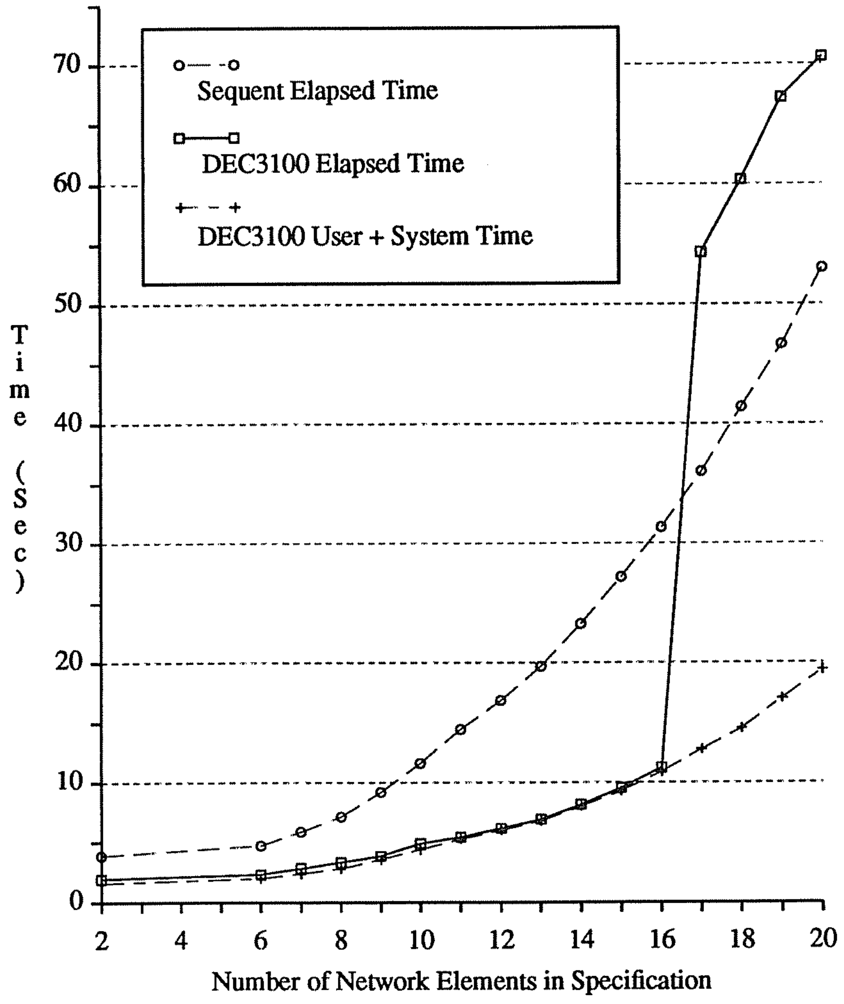

Figure 7.5.
Centralized Consistency Evaluation Time
For the Compiler Test Specification

this incongruity, continued to grow at the same rate for the tests of 17 through 20 network element specifications. This is a machine specific problem; no such increase occurred on the Sequent, with a larger main memory. This performance is bad, as we discuss in Section 7.2.2, not because the problem is difficult, but because of the CLP(R)-based implementation.

Figure 7.6 summarizes the measurements of memory utilization during centralized evaluation. We also used the data set from the Compiler Test output for these measurements. This graph shows the results for both the Sequent Symmetry and the DECstation 3100. As with evaluation time, memory use increases at a rate faster than linear. The shape of the curve is similar to the curve shown for evaluation time. This is because each time

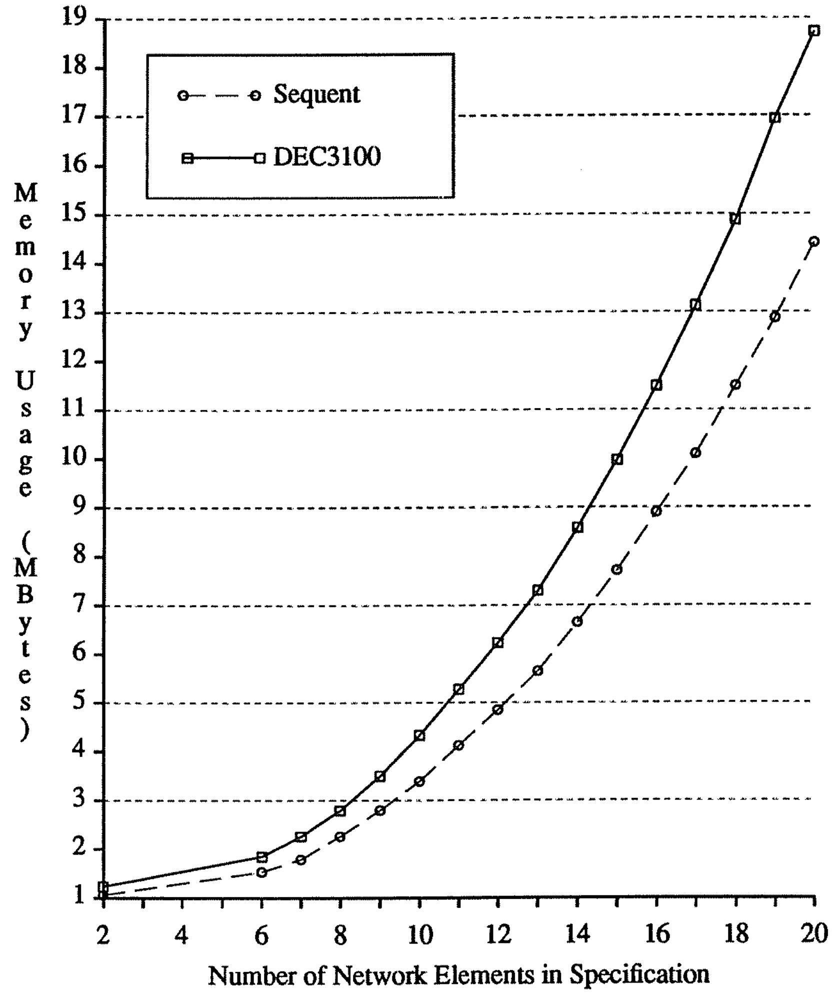

Figure 7.6.
Centralized Consistency Evaluation Memory Usage
For the Compiler Test Specification

CLP(R) evaluates a rule, it adds more information to its proof data structure. No memory is released until the proof completes.

Next, we evaluated the combined specifications for the NMSLTEST, WISCNET and CICNET domains. This test measures the centralized evaluation for specifications of a real, multi-administrative domain management system. Figure 7.7 shows the elapsed time, combined user and system time, and memory usage to check the three-domain data set using the centralized Evaluator running on the Sequent. The large difference between the elapsed time and the user and system times is due to paging and swapping activity.

Elapsed Time (min:sec)	81:44.33
User + System (min:sec)	9:09.46
Memory Usage (Mbytes)	167

Figure 7.7.
Centralized Evaluation of the Three-Domain Data Set

Next, we present the performance results for incremental evaluation. Figure 7.8 shows the input and output sizes for test specifications run through the Externalizer. We tested the Externalizer on four inputs, the NMSLTEST specification, the WISCNET specification, the CICNET specification, and the 20 network element Compiler Test specification. We show the total input size, measured in CLP(R) clauses, and the number of input clauses denoting rate modes. The table also shows the execution time of the Externalizer for these tests. The tests were run on a DECstation 3100.

In these tests, the size of the external summary is much smaller than the input specification. This means that most of the specified interactions stay within a single domain, and that there is a small set of inter-domain interactions. The CICNET summary is empty because CICNET exports data only to itself and its subdomains. When comparing centralized and incremental evaluation, the number of clauses denoting rate modes is important, because the cost of evaluation increases at a greater than linear rate with the number of clauses denoting rate modes. If the number of these clauses increases in proportion to the size of the specification, then the time to evaluate the specification will increase rapidly. If the number of these clauses increases in proportion to the size of the specification, then the time to evaluate the specification will increase rapidly. In these tests, the number of clauses denoting rate modes varied directly with the size of the specification. Finally, the Externalizer is very fast compared to the Consistency Evaluator.

Next, we determine the time to perform an incremental consistency check. To perform this test, we use the three-domain data set. This data set includes interactions between three existing network management systems,

Specification	Input Size (CLP(R) clauses)	Number of Input Clauses for Rate Modes	Output Size (CLP(R) clauses)	Execution time (sec)
NMSLTEST	431	15	10	0.39
WISCNET	911	47	11	1.50
CICNET	722	37	0	1.01
Compiler Test (20 network elements)	643	35	8	0.71

Figure 7.8.
Comparisons of Input and Output Sizes for the Externalizer

NMSLTEST, WISCNET, and CICNET. In our specifications, CICNET administrative domain contains both the NMSLTEST and WISCNET administrative domains. This administrative layout is shown in Figure 7.9. Using the domain-based incremental consistency check, the following steps are performed. First, the specifications for NMSLTEST and WISCNET are checked for consistency. Next, external summaries are made for these same specifications. The summaries are combined with the CICNET specification and checked for consistency. CICNET is the root of our test tree, so the consistency check stops here.

Figure 7.10 summarizes the elapsed time for individual steps in the incremental consistency check. The table shows elapsed times for both the Evaluator and the Externalizer for NMSLTEST and WISCNET. The Externalizer is not run on the CICNET specification because CICNET is the root of the tree. These tests were run on the Sequent Symmetry. Note that the times in Figure 7.10 are significantly smaller than those in Figure 7.7.

Figure 7.11 compares the difference between running consistency checks using a centralized vs. an incremental method. This figure summarizes the results from Figure 7.7 and 7.10 and includes the order of execution shown along a timeline. In a real incremental proof, the evaluation of specifications for domains at the same level in the tree, like NMSLTEST and WISCNET, can be run in parallel and the evaluation is shown this

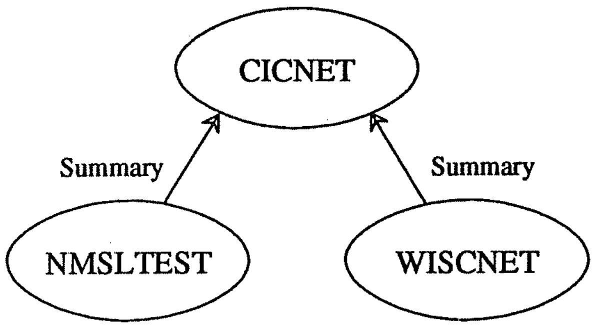

Figure 7.9.
Domain Hierarchy Used in the Incremental Tests
Arrows show propagation of summaries to the parent domain

Specification	Consistency Evaluation Elapsed Time (min:sec)	Externalizer Elapsed Time (sec)
WISCNET	2:39.8	4.1
NMSLTEST	6.3	1.1
CICNET + summaries	2:27.0	N/A

Figure 7.10.
Incremental Evaluation of the Three-Domain Data Set

way in Figure 7.11. The CICNET specification cannot be checked until it can be combined with summaries from both NMSLTEST and WISCNET. The execution time for WISCNET is longer, so the CICNET check begins when the execution of the Externalizer produces the external summary for WISCNET. The evaluation of CICNET completes about 5 minutes after the incremental evaluation begins. For comparison, the centralized check of the combined specifications for NMSLTEST, WISCNET and CICNET is shown at the bottom of the figure. It executes for a total of over 9 minutes, counting the system and user time. The parallelism accounts for only 6 seconds of this difference, so the remaining factor must be due to the smaller specifications being evaluated. Since we do not have an automated way of passing information between administrative domains, such time cannot be taken into account in these tests.

7.2.2. Discussion

The performance measurements of incremental evaluation, based on the test specifications, are encouraging. With the current Consistency Evaluator, it is much faster to execute the consistency check incrementally than it is to execute the check centrally. The incremental method runs faster because the amount of data checked in each part of the proof is smaller. Therefore, the improved speed depends on the small size of the external summaries.

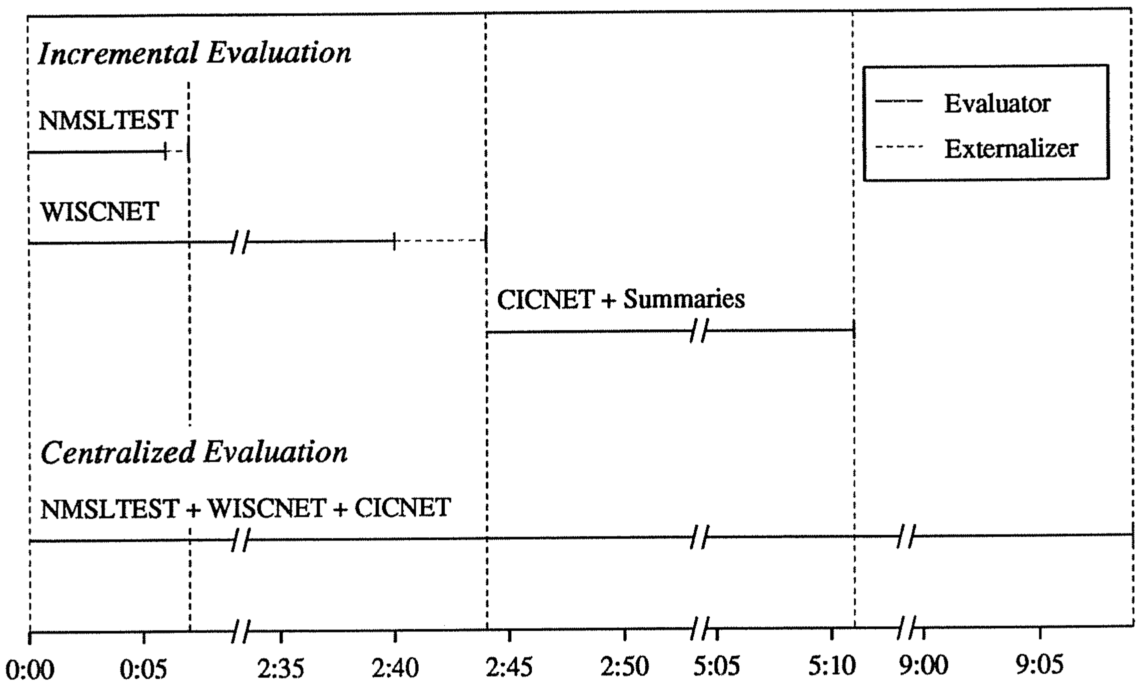

Figure 7.11.
Comparison of Incremental vs Centralized Elapsed Times
Combined User and System Time in minutes:seconds.

The data used to measure the Externalizer included only a small amount of external interactions. We preferred to use real (or realistic) specifications for the measurements, but did not find any with a large amount of external interaction. From informal inspections of other network management systems, it appears that this trend of small external interactions is common. However, we have not extensively investigated enough management systems to draw more definite conclusions.

We have investigated the effect on the incremental results of replacing the CLP(R) Evaluator with a different evaluator. We showed in Section 5.1.2 that the time to compute an aggregate query rate grows linearly with the size of the aggregate. Our test specifications indicate that the number of modes, and thus, the number of rates, grows linearly with the size of the specification. This means we need to calculate, on average, $O(N)$, aggregates, where N is the size of the specification, and implies that it takes at least $O(N^2)$ time to calculate all aggregates. This complexity results means that the best evaluator we could devise could take no less than $O(N^2)$ time to evaluate a specification. The incremental evaluation method reduces the total size of each local specification. If the sizes of the external summaries are small, as our measurements indicate, then combined summaries and parent-domain specifications will also be smaller. Depending on the constant multiplier (the time to evaluate each individual condition), our performance using the incremental method could be much better. In the current implementation, the constant is very large, so we see a large improvement. If the constant is smaller, the improvement in performance will be much lower. However, because we can perform the incremental tests in parallel, we can still achieve a faster elapsed time for the incremental proof, even for a small constant multiplier.

Our measurements of the NMSL Consistency Checker show that our choice of CLP(R), while a reasonable prototype, would need replacement to achieve acceptable performance. The main sources for the poor execution time of CLP(R) are inefficient searching algorithms and unneeded re-evaluation. Prolog implementations and CLP(R) use hashing on keys made up of the clause name and one or more parameters to reduce the search time. CLP(R) only hashes on the clause name, and NMSL uses many clauses with the same name. This means many searches down the list at each bucket need to be performed. This effect is most noticeable for the `contains` relation, used in representing the MIB object containment structure. Re-evaluation is caused by the top-down proof method used by CLP(R). As the proof moves forward, properties are proved. However, if the same property is needed again later in the proof, the property is re-proved. Some properties, such as instantiations of processes in NMSL, are re-proved many times during a CLP(R) proof. One can often use a technique called *memoization[8]* to combat this problem. Memoization simply remembers results proven (facts) within the CLP(R) fact database, and uses these results when they are re-referenced. This method did not prove useful in optimizing our implementation.

We spent many hours optimizing the CLP(R)-based evaluation rules. The main method for reducing the evaluation time was to add *cuts* to the proof when possible. A cut is used by CLP(R) to reduce backtracking during evaluation by telling the evaluator which choices need not be reconsidered during backtracking. Using cuts is tricky, because they can change the meaning of an evaluation rule if used incorrectly. By strategic placement of cuts, we were able to get a 20-fold improvement in execution time. However, cuts do not solve the basic problems of poor searching and re-evaluation.

Two alternatives to CLP(R) are to use a different logic programming language or to write a specialized evaluator. One alternative language is Coral. As we described in Chapter 5, NMSL proofs can be performed in Coral. Coral uses a bottom-up approach for proving results, rather than CLP(R)'s top-down approach. Because of the bottom-up approach, Coral avoids re-evaluation of parts of the proof. Coral was not used because a compiler was not yet available when we chose an evaluation language. A specialized evaluator could be tailored to performing consistency evaluation for NMSL specifications. This means we can optimize the algorithms used and achieve better performance than we could get from using a general purpose language. Such choices are matters for future research.

7.3. The Configuration Installer

We implemented a Configuration Installer for NMSL. It can install configurations into a running SNMP-based network management system. The system in which we use the Configuration Installer is based on the ISODE SNMP[31,32] agent and trap handler and a NYSERNET[33] monitoring process. This system, the NMSLTEST system also used in specification tests, runs on a small group of computers in the University of Wisconsin Computer Sciences Department.

To install the configuration into a running system, we use the SNMP protocol whenever possible. However, the standard MIB for SNMP does not include a common method for describing the configuration of network managers. Therefore, we added a new class of objects to our experimental SNMP system to hold configuration data. We also added a trigger object to cause reconfiguration. The object, an integer, normally has the value zero. When the object is set to one, the agent reconfigures itself, and sets the value back to zero. The agent answers no queries while it reconfigures itself. The NMSL Configuration Installer attempts to connect to each agent in the system, authenticate itself, and load the new configuration. To reduce the reconfiguration overhead, the Installer only reloads the configuration if the agent is not currently configured correctly. If any one of these steps fails, the installer tries to restart the agent (by killing the old process and re-executing the agent program). If restarting fails, the domain administrator is notified. In the last case, the running network management system is in an inconsistent state, although the specification was consistent. Recovering from such failures is one problem we plan to investigate in the future.

7.4. The Configuration Enforcer

We also implemented a very limited Configuration Enforcer. In our implementation, we only have access to the current state of agent processes. We did not have access to the runtime state of manager processes. This severely limited the amount of enforcement possible.

Agent processes respond to queries, but never initiate queries. Therefore, if the NMSL system installs the configuration in the agent, the agent should remain correctly configured unless an administrator installs a new configuration. Monitoring these agents for correct behavior consists of making sure the currently installed configuration matches the one the NMSL system installed. If the configuration is not correct, the agent is reconfigured. We can monitor the query rates, access types, and configuration requirements. Since we only have one-sided monitoring, this does not completely solve the problem of enforcing the specification on the processes initiating the queries, nor will this monitoring always tell us if an agent process is processing queries at the specified rate. For example, if too many processes are sending queries, this may not be noted in the agent's statistics, because the overload of queries might get dropped or lost in the network itself.

Enforcement of specifications was not a goal for our current research, however we do want to investigate enforcement issues in the future. To effectively enforce a specification requires all network management processes to make their state available via the management protocols for both monitoring and control. We discuss this and other issues of enforcement in our discussion of future work, in Chapter 8.

Chapter 8

Conclusion

This dissertation addressed problems caused by the increasing use of automated network management: that of managing the management system. We have described a language for specifying multi-administrative domain network management systems. We have also shown how this language is used to find consistent configurations for network management systems using an incremental proof technique. This technique preserves the autonomy of the individual domains while still allowing cooperation and controlled sharing of information between multiple domains.

We have described a specification language, NMSL, that allows network administrators to describe the configuration of their management systems and the interaction between the components. Most interactions are described in terms of the information passed between processes. Additional interactions are described in terms of the configuration requirements on the network management system. The specification includes not only the internal behavior of an administrative domain, but its external interactions as well.

We described a way to check NMSL specifications for consistency. The definition of consistency includes protection, capacity, and configuration, and allows us to ensure compliance with all requirements on interactions between components of a network management system. An important part of these specifications is the ability to place requirements on an entire domain and its interactions with other domains.

We described an incremental method for evaluating the consistency proof. We showed that we can use this incremental approach to partition the consistency proof along administrative domain boundaries. The incremental method allows us to preserve the autonomy of administrative domains. Only information pertaining to the interactions between domains needs to be shared, internal information about a domain remains private. It also has a performance advantage over evaluating the proof in a centralized manner by allowing multiple parts of the proof to be performed separately, and even simultaneously on different hosts.

We also described how one can use a consistent specification prescriptively. By automatically installing a consistent configuration into the network management system, we help to ensure that the components of the system will meet the specification. The specification can also be enforced after installation, although our investigation did not emphasize this aspect of the research.

Finally, we showed the practicality of these ideas in an implementation. The implementation concentrated on the language and consistency checking issues. We wrote specifications for existing network management systems, and showed that such specifications can be evaluated by our implementation. We provided performance measurements for both the compiler and consistency checker, and showed that the implementation works well for small network management specifications. In our implementation, the incremental proof method has an great advantage over the centralize method. We also showed that, given relatively small numbers of interactions between domains, the incremental method can always give better performance than the centralized method.

8.1. Future Work

Our investigation of NMSL suggests several areas for future research. Many of these are improvements to NMSL. Our current capacity model is very simple. There is no way to describe the length of time that a process remains in a mode of query behavior, nor is there a way to describe temporal or other cause-and-effect relationships between queries. Adding either of these to the capacity model significantly increases the complexity of the model. Simulation or evaluating a queuing model are a possible solutions for small specifications, like those presented in this dissertation, but both would be too costly for large specifications (of thousands or tens of thousands of network elements and process instances). Two possible alternative approaches to investigate are temporal logics and approximation methods that could be solved analytically. At this point, we have no feel for which alternative offers the most promise.

Another area for improvement is the enforcement of specifications. To fully enforce specifications, we need a way to monitor and control manager processes, not just agent processes. Like agent processes, manager processes should be accessed via a network management protocol. Making other processes accessible requires extensions to the management information base to describe the state of application processes. Given these extensions and a standard mechanism of locating manager processes at runtime, a specification can be enforced on all network management processes. Another issue is how to enforce a specification on processes that cannot be reconfigured, or those processes that continue to violate the specification despite being reconfigured (buggy software). Enforcement in this case would require some form of rules for penalizing uncooperative processes. These rules would generally limit a process that violates its specification and prevent it from communicating. Such limits could also violate the consistency of a specification. A specification presumably includes descriptions of the correct behavior for uncooperative processes, including interactions with other processes. However, limits would ensure a well-known behavior on processes, and limit the effect of uncooperative processes on the rest of the network management system.

We also wish to evaluate NMSL's ability to specify network management systems based on other management standards, for example, the OSI standards. Experiments with such systems will help to evaluate our language design and suggest possible areas of improvement.

The current evaluator is very slow for large specifications because CLP(R) is a general logic programming language, and is not optimized for our problem. To solve this, we want to investigate using other logic programming languages to evaluate NMSL. One possible language is CORAL. CORAL is promising because it addresses one of the main problems we found with CLP(R), re-evaluation of facts. CORAL performs proofs bottom-up and never needs to re-evaluate facts. The other main performance problem, poor searching, may also be solved by the bottom-up proof technique. We must also determine if CORAL can represent any extensions we make to the capacity model. If not, we will need to look at different languages, or write a specialized evaluator.

The Consistency Checker relies on human interaction to run the Externalizer and to send the external summary to the parent domain's administrator for use in the incremental proof. Human interaction is also needed in returning inconsistency results. Manual propagation means that the tools do not have to worry about synchronization, but it also means we cannot accurately determine the average time to perform an incremental proof. We want to automate the propagation of summaries and results. Automation requires that we deal with synchronization within a single consistency check and between proofs of different specification versions. One possible way to simplify these problems is to use a different incremental approach. One approach is to exchange external summaries between peer domains rather than relying on a domain hierarchy. This approach reduces synchronization to a pairwise problem. Additional information may also be needed in a specification to help the Evaluator and Externalizer cooperate to ensure all of the necessary data is present before checking the consistency of a specification.

We also want to extract other information from a specification, in addition to the specification's consistency. For example, if a specification is inconsistent, suggestions on how to resolve the inconsistency might be returned. Such suggestions would be based, in part, on previous inconsistency results or by comparing the differences between subsequent specification versions. For example, a common error might be to specify the modification of a read-only object. If this inconsistency is found, the suggestion can be made to inspect the offending `queries` clause, rather than checking the corresponding `exports` clause. Another possibility is to use the specification to find resources in the system that are underutilized. A specification could be consistent and could be used by a running system, but there are often economic factors that require resources (hardware, network links) to be utilized more efficiently. Both of these uses go beyond the issue of consistency evaluation.

We claimed that the language can be used not only for network management systems, but also for the more general problem of distributed systems management. The main difference between these two types of systems is the amount of information that needs to be managed. A distributed systems manager has more processes to manage and has more management processes. The structure of the language is appropriate for either management problem. However, the current evaluator cannot perform well for large specifications. We want to test NMSL on large network management and distributed management systems. Such tests are necessary to better evaluate the choices we made in our implementation and, if necessary, to re-implement parts of the NMSL system.

Finally, our work has shown the use of applying incremental approaches to management specification. We feel that incremental or distributed approaches can be used effectively in other aspects of network management. Current computer network management systems are highly centralized. Almost all network management decision-making is done by a centralized network management system. Management functionality can be distributed in large networks, and management decisions can be made incrementally. Local management processes can monitor, control and gather statistics about small parts of a network. Summaries can be passed to higher level processes. Alarms can also be handled in this way, reducing the load on centralized alarm handlers. A decentralized structure can be more robust than a centralized system, but is also more complex. Much work has been done to decentralized management in telephone networks. However, as recent problems in the long distance telephone networks show, there is still work to be done in designing robust, distributed management systems.

Appendix A

The NMSL Language Reference Manual

A.1. Introduction

This reference manual describes the implementation of the NMSL language. It is meant for programmers or administrators that wish to write NMSL specifications and who need little tutorial help. This description is bottom-up, starting at the lowest level of the syntax. Throughout, characters or symbols presented in the `typewriter` font are tokens of the language itself. The structure of the encapsulated ASN.1 notation is not discussed here, except as required in the discussion of other descriptions.

A.2. Lexical Conventions

A NMSL specification is stored in one or more files. Each file is made up of a sequence of ASCII characters. The files are first preprocessed using the C preprocessor[21]. The C preprocessor performs macro expansion. When the preprocessing phase is complete, the files have been reduced to a sequence of tokens.

There are five classes of tokens: reserved keywords, identifiers, constants, operators, and other separators. Some tokens are prefixes of other tokens. In this case, the compiler finds the maximal length valid token. Tokens are separated by whitespace. Whitespace is typically a space, tab, or newline character, and has no other meaning than to separate tokens. Some tokens, namely separators and operators, need no whitespace to differentiate them from the surrounding tokens.

There are two types of comments in NMSL. The token `/*` denotes the beginning of a comment. All tokens and characters inside a comment are ignored, until the token `*/` is seen. The other type of comment is introduced by `--` and ends with a newline. Comments do not nest.

A.2.1. Reserved Keywords

The following list of keywords are reserved and may not be used except in their specific contexts. Keywords are case-sensitive.

```
access      interface   of          seconds
avg         K           on          speed
bps         max         opsys       supports
count       Mbps        peer        system
cpu         minutes     process     to
domain      mode        provides    traps
```

enclosing	module	qps	type
end	modifies	queries	version
exports	net	rate	
external	objects	requests	
hours	octets	requires	

In addition, the following keywords, defined by ASN.1[19], are reserved:

ABSENT	END	NAMES	SIZE
ACCESS	ENUMERATED	NOTATION	STATUS
ANY	EXPLICIT	NULL	STRING
APPLICATION	EXPORTS	OBJECT	SUPERIORS
BEGIN	FALSE	OBJECT-CLASS	SYNTAX
BIT	FROM	OBJECT-TYPE	TAGS
BOOLEAN	IDENTIFIER	OCTET	TRUE
CHOICE	IMPLICIT	OF	TYPE
COMPONENT	IMPORTS	OPTIONAL	UNIVERSAL
COMPONENTS	INCLUDES	PRESENT	VALUE
CONTAINS	INTEGER	PRIVATE	WITH
DEFAULT	MACRO	REAL	
DEFINITIONS	MAX	SEQUENCE	
ENCRYPTED	MIN	SET	

A.2.2. Identifiers

An identifier is either simple or compound. A simple identifier is a sequence of letters, digits, and the _ (underscore) and - (hyphen) symbols. Identifiers must start with a letter. Case is significant. A compound identifier is a sequence of simple identifiers separated by . (period).

A.2.3. Constants

Constants come in two forms.

Constant ::= *FloatNumber* |
String

A floating point constant is a sequence of digits, optionally followed by a decimal point and a fractional part, which is a sequence of digits. Character string constants begin and end with a double quote. All characters between the quotes are part of the constant. A double quote may be made part of a string by entering two successive double quotes, e.g. `"I said, ""hi"""`.

A.2.4. Operators

Five operators are provided for describing value ranges, these operators are =, <=, <, >, >=.

A.3. Meaning of Identifiers

Identifiers are used for parameterized process declarations and for naming objects specified in ASN.1 descriptions. Process parameters are simple identifiers. They are place holders and allow one to specify constant values on a per-instance basis rather than a per-declaration basis. Parameters are all assumed to hold the place of string constants. Parameters also have a scope, which is the process declaration in which they are defined. Object names are compound identifiers. An object name identifies a kind of data stored in an instance of a network management process. Syntactically, each element in the object name is an identifier defined in an ASN.1 module and exported to other declarations. Compound identifiers allow the expression of hierarchical names, such as those used for naming objects in the SNMP MIB. For example, `mgmt.mib.system.sysUpTime` is the name of one object and `mgmt.mib.system` names all objects below `mgmt.mib.system` in the SNMP MIB naming hierarchy.

A.4. Expressions

NMSL has three types of expressions: primaries, range expressions, and object lists. Primaries are used to identify declaration names within other declarations. Ranges associate configuration conditions with a value or range of values. Object lists are simply lists of object names.

A.4.1. Primaries

A primary is either a string or a simple identifier.

primary ::= *String | Identifier*

An identifier is a primary if it is declared as a parameter within the same declaration in which it is used. Identifiers currently only identify strings.

A.4.2. Range Expressions

Range expressions appear in `provides` and `requires` clauses. Range expressions associate configuration conditions with a value or a range of values.

Range ::= *CondName |*
CondName = String |
CondName Operator FloatNumber Units |
`count` (*CondName*) *Operator FloatNumber Units*

CondName ::= *String*

Operator ::= = | < | <= | > | >=

Units ::= `K` | `hours` | `minutes` | `seconds` | `octets` | *EMPTY*

A configuration condition is a name. Syntactically, it is a character string constant. A configuration condition can be assigned a value or a range of values in a `provides` clause. A configuration condition can be tested in a `requires` clause. Specifying a `count` of a configuration condition tests the number of instances of the configuration condition rather than its value. The scope of configuration conditions are defined by the clauses that contain them.

A.4.3. Object lists

An object list is a list of object names defined in a `type` declaration. The object names are separated by commas.

VarList ::=	*VarList , compound-identifier* |
	compound-identifier

There is no semantic significance to the order in an object list. Clauses that use object lists treat all elements of the list equally.

In the `using` subclause (section A.5.2), it is necessary to associate values with object names. In this case, each element in the object list is an object name and a constant value, separated by = (an equals sign). There is no semantic significance to the order in an object list.

AsgnVarList ::=	*AsgnVarList , compound-identifier = Constant* |
	compound-identifier = Constant

A.5. Clauses

NMSL supports several clauses for describing the interactions between parts of a management system.

Clause ::=	*SupportsClause* |
	ExportsClause |
	QueriesClause |
	ProvidesClause |
	RequiresClause |
	CpuClause |
	InterfaceClause |
	OpsysClause |
	ProcessClause |
	DomainClause |
	SystemClause

A.5.1. The Supports Clause

A `supports` clause is introduced by the `supports` keyword and lists the objects from the ASN.1 module descriptions supported in the description.

SupportsClause ::= `supports` *VarList ;*

Support of an object means the definition of this object is imported from the ASN.1 module and can be used in other clauses within the same description. `Supports` clauses are currently used only for compile time checks, not by the output generators.

A.5.2. The Exports and Queries Clauses

An `exports` clause is introduced by the `exports` keyword and lists constraints on the objects exported to some process or domain. A `queries` clause is introduced by the `queries` keyword and lists the parameters to a query, including the process that is queried, the objects read or modified, and the query rates.

ExportsClause ::= `exports to` *primary* { *ExpSbclauses* }

```
QueriesClause ::= queries primary { QrySbclauses }
ExpSbclauses ::=  ExpSbclause ExpSbclauses
ExpSbclause ::=   TrapSb | ERateSb | ObjSb | AccessSb
QrySbclauses ::=  QrySbclause QrySbclauses
QrySbclause ::=   OnSb | TrapSb | QRateSb | ReqSb | ModSb | UseSb
TrapSb ::=        traps TrapIds ;
TrapIds ::=       StringList | any
StringList ::=    String | String , Stringlist
ERateSb ::=       rate max { RtModes }
ObjSb ::=         objects VarList ;
AccessSb ::=      access AcType ;
AcType ::=        ReadWrite | ReadOnly |WriteOnly | None
OnSb ::=          on primary ;
ReqSb ::=         requests VarList ;
ModSb ::=         modifies VarList ;
UseSb ::=         using AsgnVarList ;
QRateSb ::=       rate RtType { RtModes }
RtType ::=        max | avg | Float
Rtmodes ::=       RtMode RtModes
RtMode ::=        mode Integer Float Integer qps ;
```

An `exports` clause lists the recipient of the export, which must be the name of a domain. It also lists the modes in which the exporter operates. There is typically a single mode, but more are allowed. The clause also lists the types of objects that are exported. The rate modes listed in the `rate` subclause are the maximum rate at which the exporter can receive queries without becoming overloaded. Finally, the type of access that may be performed on the exported object types are listed in the `access` subclause.

A `queries` clause lists the process that will receive the query and the host on which the receiving process is instantiated. Several subclauses are used to list the objects involved in the query. The `requests` subclause lists objects whose values are to be returned by the query. The `modifies` subclause lists objects whose values will be changed by the query. The `using` subclause lists objects whose values are to be used by the recipient of the query in satisfying the query. The object and value pairs listed in the *AsgnVarList* select particular instances of the objects listed in the `requests` and `modifies` subclauses.

Both clauses list trap types, also known as exceptions or alarms, that are exported or queried. In this case, exporting a trap means that processes are allowed to send trap messages of the named types to the exporter. A query that specifies a trap means that the trap types specified will be sent to the exporting process.

A.5.3. The Provides and Requires Clauses

A `provides` clause is introduced by the `provides` keyword and describes a single configuration condition and the value or range of values provided for this condition. A `requires` clause is introduced by the `requires` keyword and describes a single configuration condition and the value or range of values that are required of this configuration condition.

```
ProvidesClause ::= provides Range ToPart ;
RequiresClause ::= requires Range OfPart ;
ToPart ::=         to PeerPart | EMPTY
OfPart ::=         of PeerPart | EMPTY
PeerPart ::=       peer | enclosing | self
```

Specifying a range in a `provides` clause says that the configuration condition can have any value in that range. Specifying a range in a `requires` clause says that all values of the configuration condition given in `provides` clauses must be in the specified range. There is no current meaning to specifying a `count` in the range of a `provides` clause, and attempts to use `count` in a `provides` clause are flagged as errors by the compiler. Specifying a count in a `requires` clause says that the number of instances of the configuration condition must be in the specified range. There is one instance for each instantiation of a process providing a configuration condition, and one instance for each network element or domain providing a configuration condition.

The context of the provision or requirement is listed using the `to` or `of` subclauses. If the context is `peer`, the condition applies to all peers of communication. The `peer` context can only be used within process descriptions. If the context is `containing`, the condition applies to the containing context. A network element contains processes, and a domain contains domains and network elements. For example, specifying `containing` within a process description means the condition applies to all network elements that instantiate the process. If the context is `self`, the condition applies to the current description and any contexts listed within the description. For example, specifying `self` within a domain description means the condition applies to the domain description and any network elements or domains that are listed as members of that domain. If no context is specified, it defaults to `self`.

A.5.4. The Cpu and Interface Clauses

These clauses are introduced by the `cpu` or `interface` keywords and describes the cpu type and network interface characteristics, respectively.

```
CpuClause ::=       cpu String ;
InterfaceClause ::= interface String {
                    net String
                    IFAddresses
                    type String
                    speed Integer IFRate }
IFRate ::=          bps | Mbps
```

IFAddresses ::= *EMPTY* |
address *String* ; |
address *String* ; remaddress *String* ;

The name of a CPU type is a string. There is currently no standard naming convention for CPUs. The name for an interface is also a string. This name should correspond to the name used by the management system to identify the interface. On Berkeley Unix variants, these names should be the names listed by the netstat program.

The net subclause lists the network name accessed via this interface. The string should be the standard name for the network as assigned by a standards body, for example, those assigned by the Internet naming authority. If there is no naming authority for a particular management system, the names may be chosen arbitrarily, but names must be used consistently (the same name each for each reference to the same network, and different names for different networks).

The type of the interface is a string and should be a standard name if possible. For example, the SNMP MIB defines names for standard interfaces.

The address and remaddress subclauses list the address of the interface and, for point-to-point links, the address on the far end of the link. The compiler uses the network type to determine if it is appropriate to specify a remote address for the interface. For IP networks, these addresses must be the internet addresses of the interface and, for point-to-point links, the remote interface.

The speed is the nominal speed of the network accessed via this interface.

A.5.5. The Opsys Clause

An opsys clause is introduced by the opsys keyword and describes the operating system type and version for a network element.

OpsysClause ::= opsys *String* version *String* ;

A.5.6. The Process Clause

A process clause is introduced by the process keyword and describes a single process instance within a network element description.

ProcessClause ::= process *String ProcParams* ;

ProcParams ::= (*PrParamList*) | *EMPTY*

PrParamList ::= *PrParamList* , *PrParam* | *PrParam*

PrParam ::= *String*

The process clause is used to instantiate processes described with process descriptions. The name of the process must match the name of a process description found elsewhere in the specification. If that process description is parameterized, a list of values to associate with the identifiers in the description's parameter list must be present. Since identifiers are only used for strings, the values must all be strings.

A.5.7. The Domain and System Clauses

These clauses are introduced by the `domain` or `system` keywords. Each lists a single member of a domain.

DomainClause ::= `domain` *String ;*

SystemClause ::= `system` *String ;*

These clauses are used only within a domain description. They describe the containment relations between the domain being described and its members.

A.6. Descriptions

There are four kinds of descriptions in NMSL. A description is used to group all of the properties (described by clauses) of a part of a network management system. A group of descriptions makes up a specification. The order of descriptions in a specification is unimportant.

Description ::= *TypeDescription* **|**
ProcessDescription **|**
NetElemDescription **|**
DomainDescription

Specification ::= *Description* **|** *Description Specification*

Each description is identified by a name, as discussed below. The names of the descriptions must be unique within a specification.

A.6.1. Type Descriptions

A type description is used to define a management information base (MIB). More than one MIB may be present in a single specification.

TypeDescription ::= `module` *ASN1syntax* .

A type description starts with the `module` keyword. The body of a type description is an ASN.1 module. The ASN.1 module syntax is described in [19]. The `OBJECT-TYPE` and `OBJECT-CLASS` ASN.1 macros for the IETF standards[28, 35] are supported. In our implementation, new ASN.1 macros cannot be defined.

A.6.2. Process Descriptions

A process description is used to define the external behavior of a network management process.

ProcessDescription ::=	`process` *String ParamSpec* `::=` *ProcBody* `end process` *String* `.`
ProcBody ::=	*ProcClause ProcBody* | *ProcClause*
ProcClause ::=	*SupportsClause* | *ExportsClause* | *QueriesClause* | *ProvidesClause* | *RequiresClause*
ParamSpec ::=	(*ParamList*) | *EMPTY*
ParamList ::=	*ParamList* , *Identifier* | *Identifier*

Any MIB objects used in the specification must be imported by including a `supports` clause. The behavior of the process is described in terms of data exported, using the `exports` clause, queries made on other processes, using the `queries` clause, and configuration constraints, using the `provides` and `requires` clauses.

Process descriptions can be parameterized with a list of identifiers. When the process is instantiated, a value must be specified for each identifier. Currently, all identifiers are assumed to identify string constants. In the future, they may be used for identifying floating-point constants as well. Identifiers are valid for the scope of the process description. Their names must be unique within a single process description, but the same name may be used again in another process description. When a process is instantiated, the values listed are used in place of the identifiers for consistency and configuration output.

A.6.3. Network Element Descriptions

A network element describes a hardware component attached to the network, such as a computer, router, or gateway, but not the wires over which data passes.

NetElemDescription ::=	`system` *String* `::=` *SysBody* `end system` *String* `.`
SysBody ::=	*SysClause SysBody* | *SysClause*
SysClause ::=	*CpuClause* | *InterfaceClause* | *OpsysClause* | *ProvidesClause* | *RequiresClause* | *ProcessClause*

A device is described in terms of the interface to the network, using the `interface` clause and the type of cpu, with the `cpu` clause. The type of operating system is also specified, using the `opsys` clause. Configuration conditions can be specified for a network element using the `provides` and `requires` clauses.

Network elements also instantiate processes, using the `process` clause. If the corresponding `process` description is parameterized, values must be specified for the parameters. These values are matched left-to-right with the identifiers in the `process` description. Instantiating a process implies the instantiation of that process's exports, queries, and configuration conditions. It is important to remember this because the aggregate query rates depend on the number of process instances. The `count` of a configuration condition also depends on the number of process instances for a condition specified within a `process` description.

A.6.4. Domain Descriptions

Domain descriptions are used to group network elements and domains and to specify domain-wide requirements on the interactions within a domain and between domains.

DomainDescription ::= `domain` *String* `::=`
  DomBody
  `end domain` *String* `.`

DomBody ::= *DomClause DomBody*

DomClause ::= *DomainClause* | *SystemClause* | *ExportsClause* |
  ProvidesClause | *RequiresClause*

The members of a domain are listed using the `system` and `domain` clauses. A domain description can restrict the export of data from the domain to other domains by including a `exports` clause. A domain can make additional constraints on the configuration within the domain or on its containing domain by including `provides` and `requires` clauses.

A.7. Grammar

Following is a concise list of the NMSL grammar, using the BNF-style notation used throughout this language reference manual.

Specification ::= *Description* | *Description Specification*

Description ::= *TypeDescription* |
  ProcessDescription |
  NetElemDescription |
  DomainDescription

TypeDescription ::= `module` *ASN1syntax* `.`

ProcessDescription ::= `process` *String ParamSpec* `::=`
  ProcBody
  `end process` *String* `.`

ProcBody ::= *ProcClause ProcBody* | *ProcClause*

ProcClause ::= *SupportsClause* | *ExportsClause* | *QueriesClause* |
  ProvidesClause | *RequiresClause*

ParamSpec ::= (*ParamList*) | *EMPTY*

ParamList ::= *ParamList* , *Identifier* | *Identifier*

NetElemDescription ::= `system` *String* `::=`
  SysBody
  `end system` *String* `.`

SysBody ::= *SysClause SysBody* | *SysClause*

SysClause ::= *CpuClause* | *InterfaceClause* | *OpsysClause* |
  ProvidesClause | *RequiresClause* | *ProcessClause*

DomainDescription ::=`domain` *String* `::=`
  DomBody
  `end domain` *String* `.`

DomBody ::= *DomClause DomBody*

```
DomClause ::=         DomainClause | SystemClause | ExportsClause |
                      ProvidesClause | RequiresClause
SupportsClause ::=    supports VarList ;
ExportsClause ::=     exports to primary { ExpSbclauses }
QueriesClause ::=     queries primary { QrySbclauses }
ExpSbclauses ::=      ExpSbclause ExpSbclauses
ExpSbclause ::=       TrapSb | ERateSb | ObjSb | AccessSb
QrySbclauses ::=      QrySbclause QrySbclauses
QrySbclause ::=       OnSb | TrapSb | QRateSb | ReqSb | ModSb | UseSb
TrapSb ::=            traps TrapIds ;
TrapIds ::=           StringList | any
StringList ::=        String | String , Stringlist
ERateSb ::=           rate max { RtModes }
ObjSb ::=             objects VarList ;
AccessSb ::=          access AcType ;
AcType ::=            ReadWrite | ReadOnly |WriteOnly | None
OnSb ::=              on primary ;
ReqSb ::=             requests VarList ;
ModSb ::=             modifies VarList ;
UseSb ::=             using AsgnVarList ;
QRateSb ::=           rate RtType { RtModes }
RtType ::=            max | avg | Float
Rtmodes ::=           RtMode  RtModes
RtMode ::=            mode Integer Float Integer qps ;
ProvidesClause ::=    provides Range ToPart ;
RequiresClause ::=    requires Range OfPart ;
ToPart ::=            to PeerPart | EMPTY
OfPart ::=            of PeerPart | EMPTY
PeerPart ::=          peer | enclosing | self
CpuClause ::=         cpu String ;
InterfaceClause ::=   interface String {
                      net String
                      IFAddresses
                      type String
                      speed Integer IFRate }
IFRate ::=            bps | Mbps
IFAddresses ::=       EMPTY |
                      address String ; |
                      address String ; remaddress String ;
OpsysClause ::=       opsys String version String ;
```

ProcessClause ::=	`process` *String ProcParams ;*
ProcParams ::=	(*PrParamList*) | *EMPTY*
PrParamList ::=	*PrParamList , PrParam* | *PrParam*
PrParam ::=	*String*
DomainClause ::=	`domain` *String ;*
SystemClause ::=	`system` *String ;*
VarList ::=	*VarList , compound-identifier* | *compound-identifier*
AsgnVarList ::=	*AsgnVarList , compound-identifier = Constant* | *compound-identifier = Constant*
Range ::=	*CondName* | *CondName = String* | *CondName Operator FloatNumber Units* | `count` (*CondName*) *Operator FloatNumber Units*
CondName ::=	*String*
Operator ::=	= | < | <= | > | >=
Units ::=	`K` | `hours` | `minutes` | `seconds` | `octets` | *EMPTY*
primary ::=	*String* | *Identifier*
Constant ::=	*FloatNumber* | *String*

Appendix B

The NMSLTEST Specification

This appendix is divided into two parts. The first section is the specification for the NMSLTEST system, described in Chapter 7. Parts of this specification are also used in examples in Chapter 5. The second section shows the CLP(R) representation of the NMSLTEST specification, as output by the NMSL compiler. The initial part of the CLP(R) representation is for the MIB and has been abbreviated here. The representation of the process, network element and domain descriptions follow these object descriptions. The text of the MIB is not included in this appendix. The reader may refer to RFC1156[27] for the full text of the MIB.

B.1. The NMSL Specification for NMSLTEST

```
#include "tests/smi"
#include "tests/mib"

process "snmpdReadOnly" ::=
  supports mgmt.mib; -- entire MIB subtree

  provides "protocol" = "snmp" to peer;
  requires "protocol" = "udp" of enclosing;
  exports to "monitor" {
    objects mgmt.mib.system,mgmt.mib.at;
    access ReadOnly;
    rate max {
      mode 1 1.00 10 qps;
    }
  }
  exports to "wisc-cs" {
    objects mgmt.mib;
    access ReadOnly;
    rate max {
      mode 1 1.00 10 qps;
    }
  }
end process "snmpdReadOnly".

process "snmpdNoEgpReadOnly" ::=
  supports
    mgmt.mib.system, mgmt.mib.at,
    mgmt.mib.interfaces,
    mgmt.mib.ip, mgmt.mib.icmp,
    mgmt.mib.tcp, mgmt.mib.udp;

  provides "isode";
  requires "protocol" = "udp" of enclosing;
  provides "protocol" = "snmp" to peer;
  queries "snmpt" {
    on "romano";
    traps any;
    rate max {
        mode 1 0.01 1 qps;
        mode 2 0.99 0 qps;
    }
  }
  exports to "wisc-cs" {
    objects mgmt.mib;
    access ReadWrite;
    rate max {
        mode 1 1.00 10 qps;
    }
  }
end process "snmpdNoEgpReadOnly".

process "snmpt" ::=
  requires "protocol" = "udp" of enclosing;
  provides "protocol" = "snmp" to peer;
  provides "isode-traps";

  exports to "wisc-cs" {
    traps any;
    rate max {
        mode 1 1.00 10 qps;
    }
  }
end process "snmpt".

process "snmpxmon" ::=
  supports mgmt.mib; -- entire MIB subtree
  provides "protocol" = "snmp" to peer;
  requires "protocol" = "snmp" of peer;
  requires "protocol" = "udp" of enclosing;
  queries "snmpdReadOnly" {
    on "gw";
    requests mgmt.mib.interface.ifTable.ifEntry.ifOperStatus;
    rate avg { mode 1 0.02 1 qps; mode 1 0.98 0 qps; }
  }
  queries "snmpdNoEgpReadOnly" {
    on "romano";
    requests mgmt.mib.interface.ifTable.ifEntry.ifOperStatus;
    rate avg { mode 1 0.02 1 qps; mode 1 0.98 0 qps; }
  }
  queries "snmpdNoEgpReadOnly" {
    on "gruyere";
    requests mgmt.mib.interface.ifTable.ifEntry.ifOperStatus;
    rate avg { mode 1 0.02 1 qps; mode 1 0.98 0 qps; }
  }
  queries "snmpdNoEgpReadOnly" {
    on "asiago";
    requests mgmt.mib.interface.ifTable.ifEntry.ifOperStatus;
    rate avg { mode 1 0.02 1 qps; mode 1 0.98 0 qps; }
  }
  queries "snmpdNoEgpReadOnly" {
    on "poona";
    requests mgmt.mib.interface.ifTable.ifEntry.ifOperStatus;
    rate avg { mode 1 0.02 1 qps; mode 1 0.98 0 qps; }
  }
  queries "snmpdNoEgpReadOnly" {
    on "beaufort";
    requests mgmt.mib.interface.ifTable.ifEntry.ifOperStatus;
    rate avg { mode 1 0.02 1 qps; mode 1 0.98 0 qps; }
  }
  queries "snmpdNoEgpReadOnly" {
    on "grilled";
    requests mgmt.mib.interface.ifTable.ifEntry.ifOperStatus;
    rate avg { mode 1 0.02 1 qps; mode 1 0.98 0 qps; }
  }
end process "snmpxmon".

#define DEC_SYS(Name,Contact,Loc)                                   \
        system Name ::=                                             \
            cpu decstation;                                         \
            interface ln0 {                                         \
                net wisc-research;                                  \
                type ethernet-csmacd;                               \
                speed 10000000 bps;                                 \
            }                                                       \
            opsys ultrix version 4.0;                               \
            process "snmpdNoEgpReadOnly";                           \
            provides "sysContact" = Contact;                        \
            provides "sysLocation" = Loc;                           \
            provides "protocol" = "udp";                            \
        end system Name .

system "gw" ::=
    cpu cisco;
    interface one {
        net cicnet;
        type T1;
        speed 1000000 bps;
    }
    interface two {
        net wisc-research;
        type ethernet-csmacd;
        speed 10000000 bps;
    }
    interface three {
        net wisc-herd;
        type ethernet-csmacd;
        speed 10000000 bps;
    }
    opsys cicso version 2;
    process "snmpdReadOnly";
    provides "protocol" = "udp";
end system "gw".

system "romano" ::=
    cpu decstation;
    interface ln0 {
        net wisc-research;
        type ethernet-csmacd;
        speed 10000000 bps;
    }
    opsys ultrix version 4.0;
    process "snmpt";
    process "snmpdNoEgpReadOnly";
    process "snmpxmon";
    provides "sysContact" = "Dave Cohrs <dave@cs.wisc.edu>";
    provides "sysLocation" = "6372 Comp Sci";
    provides "protocol" = "udp";
end system "romano" .

DEC_SYS("asiago", "Bart Miller <bart@cs.wisc.edu>",
        "6381 Comp Sci")
DEC_SYS("poona", "Jeff Hollingsworth <hollings@cs.wisc.edu>",
        "6376 Comp Sci")
DEC_SYS("beaufort", "R Bruce Irvin <rbi@cs.wisc.edu>",
        "6376 Comp Sci")
DEC_SYS("grilled", "Rob Netzer <netzer@cs.wisc.edu>",
        "6358 Comp Sci")
DEC_SYS("gruyere", "Joann Ordille <joann@cs.wisc.edu>",
        "6372 Comp Sci")

domain wisc-cs ::=
    system gw;
    system asiago;
    system romano;
    system gruyere;
    system poona;
    system beaufort;
    system grilled;
end domain wisc-cs.

domain monitor ::=
    external;
    system gw;
end domain monitor.
```

B.2. The CLP(R) Representation of the NMSLTEST Specification

```
contains([[1,3,6,1,2,1,1],[1,3,6,1,2,1,1,1]]).
decl_access([1,3,6,1,2,1,1,1],readOnly).
contains([[1,3,6,1,2,1,1],[1,3,6,1,2,1,1,2]]).
decl_access([1,3,6,1,2,1,1,2],readOnly).
contains([[1,3,6,1,2,1,1],[1,3,6,1,2,1,1,3]]).
decl_access([1,3,6,1,2,1,1,3],readOnly).
contains([[1,3,6,1,2,1,2],[1,3,6,1,2,1,2,1]]).
decl_access([1,3,6,1,2,1,2,1],readOnly).
contains([[1,3,6,1,2,1,2],[1,3,6,1,2,1,2,2]]).
decl_access([1,3,6,1,2,1,2,2],readWrite).
contains([[1,3,6,1,2,1,2,2],[1,3,6,1,2,1,2,2,1]]).
decl_access([1,3,6,1,2,1,2,2,1],readWrite).
        .
        .
        .
```

```
contains([[1,3,6,1,2,1,8,5,1],[1,3,6,1,2,1,8,5,1,2]]).
decl_access([1,3,6,1,2,1,8,5,1,2],readOnly).
instan(P1,[1,3,6,1,2,1],[P1,1,3,6,1,2,1]) :-
        instanD(S1,process('snmpdReadOnly'),P1).
provides(Y2, 'protocol', '=', 'snmp', X) :-
        instanD(S1,process('snmpdReadOnly'),Y2),
        queries(X,Y2).
requires(Y2, 'protocol', '=', 'udp', X) :-
        instanD(S1,process('snmpdReadOnly'),X),
        Y2 = S1.
perm_eq('monitor',Y,1.00,10) :-
        instanD(S1,process('snmpdReadOnly'),S2),
        instan(S2,[1,3,6,1,2,1,1 | V1],Y).
perm_access(S2,'monitor',Y,readOnly) :-
        instanD(S1,process('snmpdReadOnly'),S2),
        instan(S2,[1,3,6,1,2,1,1 | V1],Y).
perm_eq('monitor',Y,1.00,10) :-
        instanD(S1,process('snmpdReadOnly'),S2),
        instan(S2,[1,3,6,1,2,1,3|V1],Y).
perm_access(S2,'monitor',Y,readOnly) :-
        instanD(S1,process('snmpdReadOnly'),S2),
        instan(S2,[1,3,6,1,2,1,3|V1],Y).
perm_eq('wisc-cs',Y,1.00,10) :-
        instanD(S1,process('snmpdReadOnly'),S2),
        instan(S2,[1,3,6,1,2,1 | V1],Y).
perm_access(S2,'wisc-cs',Y,readOnly) :-
        instanD(S1,process('snmpdReadOnly'),S2),
        instan(S2,[1,3,6,1,2,1 | V1],Y).
instan(P1,[1,3,6,1,2,1,1],[P1,1,3,6,1,2,1,1]) :-
        instanD(S1,process('snmpdNoEgpReadOnly'),P1).
instan(P1,[1,3,6,1,2,1,3],[P1,1,3,6,1,2,1,3]) :-
        instanD(S1,process('snmpdNoEgpReadOnly'),P1).
instan(P1,[1,3,6,1,2,1,2],[P1,1,3,6,1,2,1,2]) :-
        instanD(S1,process('snmpdNoEgpReadOnly'),P1).
instan(P1,[1,3,6,1,2,1,4],[P1,1,3,6,1,2,1,4]) :-
        instanD(S1,process('snmpdNoEgpReadOnly'),P1).
instan(P1,[1,3,6,1,2,1,5],[P1,1,3,6,1,2,1,5]) :-
        instanD(S1,process('snmpdNoEgpReadOnly'),P1).
instan(P1,[1,3,6,1,2,1,6],[P1,1,3,6,1,2,1,6]) :-
        instanD(S1,process('snmpdNoEgpReadOnly'),P1).
instan(P1,[1,3,6,1,2,1,7],[P1,1,3,6,1,2,1,7]) :-
        instanD(S1,process('snmpdNoEgpReadOnly'),P1).
provides(Y2, 'isode', 'noop', '%s', X) :-
        instanD(S1,process('snmpdNoEgpReadOnly'),Y2),
        (containsD(Y2, X) ; X = Y2).
requires(Y2, 'protocol', '=', 'udp', X) :-
        instanD(S1,process('snmpdNoEgpReadOnly'),X),
        Y2 = S1.
provides(Y2, 'protocol', '=', 'snmp', X) :-
        instanD(S1,process('snmpdNoEgpReadOnly'),Y2),
        queries(X,Y2).
ref_eq(P2,Y1,0.01,1,avg) :-
        instanD(S2,process('snmpdNoEgpReadOnly'),P2),
        instanD('romano',process('snmpt'),P1),
        (
         Y1 = [traps,any]).
```

```
ref_eq(P2,Y1,0.99,0,avg) :-
        instanD(S2,process('snmpdNoEgpReadOnly'),P2),
        instanD('romano',process('snmpt'),P1),
        (
         Y1 = [traps,any]).
queries(P1,P2) :-
        instanD(S1,process('snmpdNoEgpReadOnly'),P1),
        instanD('romano',process('snmpt'),P2).
ref_access(P2,Y1,readOnly) :-
        instanD(S2,process('snmpdNoEgpReadOnly'),P2),
        instanD('romano',process('snmpt'),P1),
        (
         Y1 = [traps,any]).
perm_eq('wisc-cs',Y,1.00,10) :-
        instanD(S1,process('snmpdNoEgpReadOnly'),S2),
        instan(S2,[1,3,6,1,2,1 | V1],Y).
perm_access(S2,'wisc-cs',Y,readWrite) :-
        instanD(S1,process('snmpdNoEgpReadOnly'),S2),
        instan(S2,[1,3,6,1,2,1 | V1],Y).
requires(Y2, 'protocol', '=', 'udp', X) :-
        instanD(S1,process('snmpt'),X),
        Y2 = S1.
provides(Y2, 'protocol', '=', 'snmp', X) :-
        instanD(S1,process('snmpt'),Y2),
        queries(X,Y2).
provides(Y2, 'isode-traps', 'noop', '%s', X) :-
        instanD(S1,process('snmpt'),Y2),
        (containsD(Y2, X) ; X = Y2).
perm_eq('wisc-cs',[traps,any],1.00,10) :-
        instanD(S1,process('snmpt'),S2).
perm_access(S2,'wisc-cs',[traps,any],readOnly) :-
        instanD(S1,process('snmpt'),S2).
provides(Y2, 'protocol', '=', 'snmp', X) :-
        instanD(S1,process('snmpxmon'),Y2),
        queries(X,Y2).
requires(Y2, 'protocol', '=', 'snmp', X) :-
        instanD(S1,process('snmpxmon'),X),
        queries(X,Y2).
requires(Y2, 'protocol', '=', 'udp', X) :-
        instanD(S1,process('snmpxmon'),X),
        Y2 = S1.
ref_eq(P2,Y1,0.02,1,avg) :-
        instanD(S2,process('snmpxmon'),P2),
        instanD('gw',process('snmpdReadOnly'),P1),
        (
         instan(P1,[1,3,6,1,2,1,2,2,1,8],Y1)).
ref_eq(P2,Y1,0.98,0,avg) :-
        instanD(S2,process('snmpxmon'),P2),
        instanD('gw',process('snmpdReadOnly'),P1),
        (
         instan(P1,[1,3,6,1,2,1,2,2,1,8],Y1)).
queries(P1,P2) :-
        instanD(S1,process('snmpxmon'),P1),
        instanD('gw',process('snmpdReadOnly'),P2).
ref_access(P2,Y1,readOnly) :-
        instanD(S2,process('snmpxmon'),P2),
        instanD('gw',process('snmpdReadOnly'),P1),
        (
         instan(P1,[1,3,6,1,2,1,2,2,1,8],Y1)).
ref_eq(P2,Y1,0.02,1,avg) :-
        instanD(S2,process('snmpxmon'),P2),
        instanD('romano',process('snmpdReadOnly'),P1),
        (
         instan(P1,[1,3,6,1,2,1,2,2,1,8],Y1)).
ref_eq(P2,Y1,0.98,0,avg) :-
        instanD(S2,process('snmpxmon'),P2),
        instanD('romano',process('snmpdReadOnly'),P1),
        (
         instan(P1,[1,3,6,1,2,1,2,2,1,8],Y1)).
queries(P1,P2) :-
        instanD(S1,process('snmpxmon'),P1),
        instanD('romano',process('snmpdReadOnly'),P2).
ref_access(P2,Y1,readOnly) :-
        instanD(S2,process('snmpxmon'),P2),
        instanD('romano',process('snmpdReadOnly'),P1),
        (
         instan(P1,[1,3,6,1,2,1,2,2,1,8],Y1)).
ref_eq(P2,Y1,0.02,1,avg) :-
        instanD(S2,process('snmpxmon'),P2),
        instanD('gruyere',process('snmpdReadOnly'),P1),
        (
         instan(P1,[1,3,6,1,2,1,2,2,1,8],Y1)).
ref_eq(P2,Y1,0.98,0,avg) :-
        instanD(S2,process('snmpxmon'),P2),
        instanD('gruyere',process('snmpdReadOnly'),P1),
        (
         instan(P1,[1,3,6,1,2,1,2,2,1,8],Y1)).
queries(P1,P2) :-
        instanD(S1,process('snmpxmon'),P1),
        instanD('gruyere',process('snmpdReadOnly'),P2).
ref_access(P2,Y1,readOnly) :-
        instanD(S2,process('snmpxmon'),P2),
        instanD('gruyere',process('snmpdReadOnly'),P1),
        (
         instan(P1,[1,3,6,1,2,1,2,2,1,8],Y1)).
ref_eq(P2,Y1,0.02,1,avg) :-
        instanD(S2,process('snmpxmon'),P2),
        instanD('asiago',process('snmpdReadOnly'),P1),
        (
         instan(P1,[1,3,6,1,2,1,2,2,1,8],Y1)).
ref_eq(P2,Y1,0.98,0,avg) :-
        instanD(S2,process('snmpxmon'),P2),
        instanD('asiago',process('snmpdReadOnly'),P1),
        (
         instan(P1,[1,3,6,1,2,1,2,2,1,8],Y1)).
queries(P1,P2) :-
        instanD(S1,process('snmpxmon'),P1),
        instanD('asiago',process('snmpdReadOnly'),P2).
ref_access(P2,Y1,readOnly) :-
        instanD(S2,process('snmpxmon'),P2),
        instanD('asiago',process('snmpdReadOnly'),P1),
        (
         instan(P1,[1,3,6,1,2,1,2,2,1,8],Y1)).
ref_eq(P2,Y1,0.02,1,avg) :-
        instanD(S2,process('snmpxmon'),P2),
        instanD('poona',process('snmpdReadOnly'),P1),
        (
         instan(P1,[1,3,6,1,2,1,2,2,1,8],Y1)).
ref_eq(P2,Y1,0.98,0,avg) :-
        instanD(S2,process('snmpxmon'),P2),
        instanD('poona',process('snmpdReadOnly'),P1),
        (
         instan(P1,[1,3,6,1,2,1,2,2,1,8],Y1)).
queries(P1,P2) :-
        instanD(S1,process('snmpxmon'),P1),
        instanD('poona',process('snmpdReadOnly'),P2).
ref_access(P2,Y1,readOnly) :-
        instanD(S2,process('snmpxmon'),P2),
        instanD('poona',process('snmpdReadOnly'),P1),
        (
         instan(P1,[1,3,6,1,2,1,2,2,1,8],Y1)).
ref_eq(P2,Y1,0.02,1,avg) :-
        instanD(S2,process('snmpxmon'),P2),
        instanD('beaufort',process('snmpdReadOnly'),P1),
        (
         instan(P1,[1,3,6,1,2,1,2,2,1,8],Y1)).
ref_eq(P2,Y1,0.98,0,avg) :-
        instanD(S2,process('snmpxmon'),P2),
        instanD('beaufort',process('snmpdReadOnly'),P1),
        (
         instan(P1,[1,3,6,1,2,1,2,2,1,8],Y1)).
queries(P1,P2) :-
        instanD(S1,process('snmpxmon'),P1),
        instanD('beaufort',process('snmpdReadOnly'),P2).
ref_access(P2,Y1,readOnly) :-
        instanD(S2,process('snmpxmon'),P2),
        instanD('beaufort',process('snmpdReadOnly'),P1),
        (
         instan(P1,[1,3,6,1,2,1,2,2,1,8],Y1)).
ref_eq(P2,Y1,0.02,1,avg) :-
        instanD(S2,process('snmpxmon'),P2),
        instanD('grilled',process('snmpdReadOnly'),P1),
        (
         instan(P1,[1,3,6,1,2,1,2,2,1,8],Y1)).
ref_eq(P2,Y1,0.98,0,avg) :-
        instanD(S2,process('snmpxmon'),P2),
        instanD('grilled',process('snmpdReadOnly'),P1),
        (
         instan(P1,[1,3,6,1,2,1,2,2,1,8],Y1)).
queries(P1,P2) :-
        instanD(S1,process('snmpxmon'),P1),
        instanD('grilled',process('snmpdReadOnly'),P2).
ref_access(P2,Y1,readOnly) :-
        instanD(S2,process('snmpxmon'),P2),
        instanD('grilled',process('snmpdReadOnly'),P1),
        (
         instan(P1,[1,3,6,1,2,1,2,2,1,8],Y1)).
property('gw','cpu','cisco').
property('gw','net','cicnet').
property('gw','type','T1').
property('gw','speed',1000000).
property('gw','net','wisc-research').
property('gw','type','ethernet-csmacd').
property('gw','speed',10000000).
property('gw','net','wisc-herd').
property('gw','type','ethernet-csmacd').
property('gw','speed',10000000).
property('gw','opsys','cicso').
instanD('gw',process('snmpdReadOnly'),['gw',snmpdReadOnly]).
provides(Y2, 'protocol', '=', 'udp', X) :-
        Y2 = 'gw',
        (instanD(Y2,_,X) ; X = Y2).
property('romano','cpu','decstation').
property('romano','net','wisc-research').
property('romano','type','ethernet-csmacd').
property('romano','speed',10000000).
property('romano','opsys','ultrix').
instanD('romano',process('snmpt'),['romano',snmpt]).
instanD('romano',process('snmpdNoEgpReadOnly'),
['romano',snmpdNoEgpReadOnly]).
instanD('romano',process('snmpxmon'),['romano',snmpxmon]).
provides(Y2, 'sysContact', '=',
        'Dave Cohrs <dave@cs.wisc.edu>', X) :-
        Y2 = 'romano',
        (instanD(Y2,_,X) ; X = Y2).
provides(Y2, 'sysLocation', '=', '6372 Comp Sci', X) :-
        Y2 = 'romano',
        (instanD(Y2,_,X) ; X = Y2).
provides(Y2, 'protocol', '=', 'udp', X) :-
        Y2 = 'romano',
        (instanD(Y2,_,X) ; X = Y2).
property('asiago','cpu','decstation').
property('asiago','net','wisc-research').
property('asiago','type','ethernet-csmacd').
property('asiago','speed',10000000).
property('asiago','opsys','ultrix').
instanD('asiago',process('snmpdNoEgpReadOnly'),
        ['asiago',snmpdNoEgpReadOnly]).
provides(Y2, 'sysContact', '=',
        'Bart Miller <bart@cs.wisc.edu>', X) :-
        Y2 = 'asiago',
        (instanD(Y2,_,X) ; X = Y2).
provides(Y2, 'sysLocation', '=', '6381 Comp Sci', X) :-
        Y2 = 'asiago',
        (instanD(Y2,_,X) ; X = Y2).
provides(Y2, 'protocol', '=', 'udp', X) :-
        Y2 = 'asiago',
        (instanD(Y2,_,X) ; X = Y2).
property('poona','cpu','decstation').
property('poona','net','wisc-research').
property('poona','type','ethernet-csmacd').
property('poona','speed',10000000).
property('poona','opsys','ultrix').
instanD('poona',process('snmpdNoEgpReadOnly'),
        ['poona',snmpdNoEgpReadOnly]).
provides(Y2, 'sysContact', '=',
        'Jeff Hollingsworth <hollings@cs.wisc.edu>', X) :-
        Y2 = 'poona',
        (instanD(Y2,_,X) ; X = Y2).
provides(Y2, 'sysLocation', '=', '6376 Comp Sci', X) :-
        Y2 = 'poona',
        (instanD(Y2,_,X) ; X = Y2).
provides(Y2, 'protocol', '=', 'udp', X) :-
        Y2 = 'poona',
        (instanD(Y2,_,X) ; X = Y2).
property('beaufort','cpu','decstation').
property('beaufort','net','wisc-research').
property('beaufort','type','ethernet-csmacd').
property('beaufort','speed',10000000).
property('beaufort','opsys','ultrix').
instanD('beaufort',process('snmpdNoEgpReadOnly'),
        ['beaufort',snmpdNoEgpReadOnly]).
provides(Y2, 'sysContact', '=',
        'R Bruce Irvin <rbi@cs.wisc.edu>', X) :-
        Y2 = 'beaufort',
        (instanD(Y2,_,X) ; X = Y2).
provides(Y2, 'sysLocation', '=', '6376 Comp Sci', X) :-
        Y2 = 'beaufort',
        (instanD(Y2,_,X) ; X = Y2).
provides(Y2, 'protocol', '=', 'udp', X) :-
        Y2 = 'beaufort',
        (instanD(Y2,_,X) ; X = Y2).
property('grilled','cpu','decstation').
property('grilled','net','wisc-research').
property('grilled','type','ethernet-csmacd').
property('grilled','speed',10000000).
property('grilled','opsys','ultrix').
instanD('grilled',process('snmpdNoEgpReadOnly'),
        ['grilled',snmpdNoEgpReadOnly]).
provides(Y2, 'sysContact', '=',
        'Rob Netzer <netzer@cs.wisc.edu>', X) :-
        Y2 = 'grilled',
        (instanD(Y2,_,X) ; X = Y2).
provides(Y2, 'sysLocation', '=', '6358 Comp Sci', X) :-
        Y2 = 'grilled',
        (instanD(Y2,_,X) ; X = Y2).
provides(Y2, 'protocol', '=', 'udp', X) :-
        Y2 = 'grilled',
        (instanD(Y2,_,X) ; X = Y2).
property('gruyere','cpu','decstation').
property('gruyere','net','wisc-research').
property('gruyere','type','ethernet-csmacd').
property('gruyere','speed',10000000).
property('gruyere','opsys','ultrix').
instanD('gruyere',process('snmpdNoEgpReadOnly'),
        ['gruyere',snmpdNoEgpReadOnly]).
provides(Y2, 'sysContact', '=',
        'Joann Ordille <joann@cs.wisc.edu>', X) :-
        Y2 = 'gruyere',
        (instanD(Y2,_,X) ; X = Y2).
```

```
provides(Y2, 'sysLocation', '=', '6372 Comp Sci', X) :-
        Y2 = 'gruyere',
        (instanD(Y2,_,X) ; X = Y2).
provides(Y2, 'protocol', '=', 'udp', X) :-
        Y2 = 'gruyere',
        (instanD(Y2,_,X) ; X = Y2).
containsD(['wisc-cs','gw']).
containsD(['wisc-cs','asiago']).
containsD(['wisc-cs','romano']).
containsD(['wisc-cs','gruyere']).
containsD(['wisc-cs','poona']).
containsD(['wisc-cs','beaufort']).
containsD(['wisc-cs','grilled']).
localdomain('wisc-cs').
containsD(['monitor','gw']).
externaldomain('monitor').
```

Appendix C

The CLP(R) Representation of Evaluation Rules

```
%
% dependencies
%
?- lib(all).

%
% list generators
%
%
% contains becomes lists
%
lcontains([X,Y]) :- contains([X,Y]).
lcontains([X,Y2|Y]) :- contains([X,Y2]), lcontains([Y2|Y]).

isin2([[],_]).
isin2([[O1|O1L],[O1|O2L]]) :- isin2([O1L,O2L]).
isin([O1,O2]) :- contains([_,O2]), isin2([O1,O2]).

%
% containsD becomes lists
%
lcontainsD([X,Y]) :- containsD([X,Y]).
lcontainsD([X,Y2|Y]) :- containsD([X,Y2]), lcontainsD([Y2|Y]).

isinD2([O1,O2]) :- lcontainsD([O1|L]), tail(L,O2).

tail([X],X).
tail([X|L],Y) :- tail(L,Y).

%
% protection rules
%  - implied object and domain permissions are included in
%    compiler output
%  - uid's are implicit in the implementation;
%    no need to check for them
%
containsD([D,I]) :- instanD(D,P,I).
instan(IP,O2,[O2|IPO1]) :-  isin([O1,O2]), instan(IP,O1,IPO1), !.

%
% capacity rules
%
ref_eq(Y,[P,Prob,N,X]) :- ref_eq(P,Y,Prob,N,X).
perm_eq(A,B,C,D) :- isinD2([D1,A]), perm_eq(D1,B,C,D).
perm_eq(A,[O2|B],C,D) :- isin([O1,O2]), !, perm_eq(A,[O1|B2],C,D).
```

```
%
% verification rules, in terms of what is invalid
%

% protection condition
perm_access(S,P,Y,T) :- isinD2([D1,P]), !, perm_access(S,D1,Y,T).
perm_access(S,P,Y,T) :- nonvar(S), nonvar(P), nonvar(Y), nonvar(T),
      asserta(failed(S,P,Y,T)).

inconsistent([X2,X,Y]) :-
      ref_access(X2,Yinstan,readWrite), instan(X,Y,Yinstan),
      decl_access(Y,readOnly).
inconsistent([P,Y,T]) :- ref_access(P,Y,T), perm_access(S,P,Y,T),
      isinD2([D,S1]),not(isinD2([D,P])),perm_access(D,P,Y,T),
      failed(S1,P1,Y1,T1).
inconsistent([P,Y,T]) :-
      ref_access(P,Y,T), perm_access(S,P,Y,T), failed(S1,P1,Y1,T1).

%
% capacity condition
%

greater(A,B,A) :- A > B,A.
greater(A,B,B).

max([[[C,Prob,T,X]|Y]],T,X).
max([[[C,Prob,T,max]|Y]|Tail],Result,max) :-
      max(Tail, R2, X), greater(R2,T,Result).
max([[[C,Prob,T,X]|Y]|Tail],Result,max) :-
      max(Tail, R2, max), greater(R2,T,Result).

breaklist([[[X1|X3],X4]|[[[X1|X5],X4]|Xrest]],Y,Z) :-
      breaklist([[[X1|X5],X4]|Xrest],Y2,Z), Y = [[[X1|X3]|X4]|Y2].
breaklist([X1],[X1],[]).
breaklist([X1|X2],[X1],X2).

maxeval(_,[],[],_) :- !, fail.
maxeval([R,Result,Y,LH],[[[LH|LE],Y]|L],_,X) :-
      perm_eq(LH,Y,P,R),
      Result > R,
      max([[[LH|LE],Y]|L],Result,max).
maxeval([R,Result,Y,LH],_,L2,X) :-
      breaklist(L2, L3, L4),
      maxeval([R,Result,Y,LH],L3,L4,max).

tonum(X,X):-real(X).
tonum(X,0).

gothru([[P1,T1]],P,T,[[Pout,Tout]]) :- Pout = P * P1, Tout = T1 + T;
gothru([[P1,T1]|V],P,T,[[Pout,Tout]|Vout]):-
      Pout = P * P1, Tout = T1 + T, gothru(V,P,T,Vout);

%
% Aggregation; using the naive, exponential algorithm
% The idea: compute aggregate of Tail modes,
%       combine with current mode
%       compute aggregate for rest of modes from same client
%       concatenate the lists
% Also, find largest precentage specified, 0 if none
%
agr([[[C,Prob,T,X]|Y]],T,X) :- real(X).
agr([[[C,Prob,T,X]|Y]],T,0).
agr([],[[[C,Prob,T,X]|Y]|Tail],[Rout|R3],N) :-
      Tail = [[[C,Prob2,T2,X2]|Y2]|Tail2],
      agr(C, Tail2, R2, N1), gothru(R2,Prob,T,Rout),
      agr([], Tail, R3, N2),
      greater(N1,N2,N4), tonum(X,N3), greater(N3,N4,N).
agr([],[[[C,Prob,T,X]|Y]|Tail],Rout,N) :-
      tonum(X,N1),
      agr(C, Tail, R2, N2),
      gothru(R2,Prob,T,Rout),
      greater(N1,N2,N).
agr(C,[[[C,Prob,T,X]|Y]|Tail],R2,N) :-
      agr(C, Tail, R2, N).
agr(C2,[[[C,Prob,T,X]|Y]|Tail],Rout,N) :-
      tonum(X,N1),
      agr(C, Tail, R2, N2),
      gothru(R2,Prob,T,Rout),
      greater(N1,N2,N).

% Find %age of modes, P2, with rate greater than R
addup([[P1,T1]],R,P1) :- T1 > R.
addup([[P1,T1]],R,0).

addup([[P1,T1]|V],R,P) :-
      addup(V,R,P2), addup([[P1,T1]],R,P1), P = P1 + P2.

pct(A,Result,R,P) :-
      agr([],A,AGR,P),addup(AGR,R,P2),P > P2.

pcteval(_,[],[],_) :- !, fail.
pcteval([R,Result,Y,LH],[[[LH|LE],Y]|L],_,X) :-
      perm_eq(LH,Y,P,R),
      pct([[[LH|LE],Y]|L],Result,R,X).
pcteval([R,Result,Y,LH],_,L2,X) :-
      breaklist(L2, L3, L4),
      pcteval([R,Result,Y,LH],L3,L4,R,X).

doeval(A,B,C) :- maxeval(A,B,C,max).
doeval(A,B,C) :- pcteval(A,B,C,P).
doeval(A,B,C) :- pcteval(A,B,C,0.50).

inconsistent([LH,Result,Y,R]) :-
      retractall(unq(Xfoo)),
      all([F,X], ref_eq(X,F), List),
      breaklist(List, L1, L2),
      doeval([R,Result,Y,LH],L1,L2).

%
% Configuration conditions
%
inconsistent([I,A,OP,B,X]) :-
```

```
      requires(I,A,OP,B,X), not(provides(I,A,'=',B,X)).
inconsistent([A,'=',B]) :-
      requires(I,A,'=',B,X), not(provides(I,A,'=',B,X)).
inconsistent([A,'<',B]) :-
      requires(I,A,'<',B,X), provides(I,A,'=',B2,X), B >= B2.
inconsistent([A,'<=',B]) :-
      requires(I,A,'<=',B,X), provides(I,A,'=',B2,X), B > B2.
inconsistent([A,'>',B]) :-
      requires(I,A,'>',B,X), provides(I,A,'=',B2,X), B <= B2.
inconsistent([A,'>=',B]) :-
      requires(I,A,'>=',B,X), provides(I,A,'=',B2,X), B < B2.

sizeof([],0).
sizeof([A|B],S) :- sizeof(B,S1), S = S1 + 1;

compare(N1,N2,'=') :- N1 < N2.
compare(N1,N2,'=') :- N1 > N2.
compare(N1,N2,'<') :- N1 >= N2;
compare(N1,N2,'<=') :- N1 > N2;
compare(N1,N2,'>') :- N1 <= N2;
compare(N1,N2,'>=') :- N1 < N2;
inconsistent([A,'Count',B]) :- count(I,A,OP,B,X)
      all(A, provides(_,A,_,_,X), List),
      sizeof(List,S), compare(B,S,OP).
```

Appendix D

The NMSL Extension Language

Extension programs are divided into procedures, one for each NMSL statement. The syntax of the language is a header, defining the type of NMSL statement recognized (for example, `declhead` for the head of a declaration, or `clause` for a NMSL clause). This type is followed by an expression defining which statements of the specified type are processed using this procedure. Next, are a list of sub-procedures. The `generic` sub-procedure is used to convert the statement into an internally useful form. Other sub-procedure are used for generating output. For example, `consistency` is used to generate consistency output; `isode` is used to generate configuration output for ISODE agent processes.

The sub-procedures are made up of a list of statements. These statements, in general, should be familiar to any programmer familiar with the C programming language[21] and the csh command language[20]. The statements recognized are `foreach`, `if`, `break`, `let`, `printf`, `put`, `putprop`, `delprop`, `error`, `return`, `do`, and `dumpit`. Compound statement are also recognized and are surrounded by braces: `{`, `}`. We describe those statements that are unique to the Extension Language.

`let i = expression`

Initializes (and declares) or assigns a new value, computed from `expression` to variable `i`.

`put(key, value, ...)`

The `put` statement saves a value or an array of values in the symbol table keyed on the specified key.

`putprop(key, prop, values...)`

Similar to the `put` statement, `putprop` assigns a property named `prop` to a `key` in the symbol table with the given value(s). Many properties may be associated with the same key.

`delprop(key [,prop])`

Deletes a single property named `prop` associated with `key`, or, if `prop` is not given, deletes all properties of `key`. If `key` has a value, it's value is not changed.

`error(format, args ...)`

`Error` prints its output (the arguments are processed like a `printf` statement) to the standard error output device and aborts the compilation of the current NMSL clause. Compilation continues with the next clause, until too many errors have been found, in which case compilation is aborted.

`return(values...)`

Causes the given values to be returned from the current sub-procedure. If the sub-procedure is `generic` then these values will be the parameters of any subsequent sub-procedure instead of the original parameters. See below for more description of parameters.

`do stmt`

Loops on `stmt` (a compound statement) until a `break` statement is reached.

`dumpit(value)`

A rudimentary debugging facility. `Dumpit` completely displays the given value, even expanding arrays. Types of each element of the value are also displayed.

The `if` and `let` statements make use of expressions. The simplest expression is a single variable, constant, or function call. Simple expressions can be combined using infix operators. Operators are left-associative and all have the same precedence. Parentheses may be used to group sub-expressions. The supported operators are

`+, -, *, /`

The usual arithmetic operators. These operators only apply to numbers.

`!=, <, <=, ==, >, >=`

String and arithmetic comparators. These operators denote not-equal (!=), less than (<), less than or equal (<-), equal (==), greater than (>), and greater than or equal (>=). Strings can be compared only with != and ==. All of the comparators apply to numbers. Comparators evaluate to non-zero if the comparison is true and zero if the comparison is false.

`||, &&`

Logical-or and logical-and, respectively. Logical-or evaluates to non-zero if either argument is non-zero. Logical-and evaluates to non-zero if both arguments are non-zero. These operators evaluate to zero in all other cases.

`=~, !~`

Regular expression operators. `=~` evaluates to non-zero if the left-hand string is matched by the regular expression found in the right-hand string. It evaluates to zero in all other cases. `!~` returns the opposite result of `=~`.

Variables are not typed and can be assigned numeric, string, or array values. Functions are provided to test the type of a value. Built-in variables begin with the `$` symbol. `$decl` refers to the current declaration; `$clause` refers to the current top-level clause; `$line` refers to the current line number in the input file. `$n`, where `n` is a number, refers to parameters of the current NMSL statement. `$*` refers to the entire current NMSL statement being processed. A parameter is either a parsed token or an array of parameters. Some statements, like clauses, are made up of other statements, so parameters can be indexed. Thus, to refer to the second parameter of the third subclause of the current clause, one would use `$3[2]`.

The Extension Language provides a small set of built-in functions. Functions may appear in expressions. These include:

`get(x)`

Retrieves a value from the symbol table previously stored by the `put` statement with the name `x`. `x` is a string.

`getprop(x,y)`

Retrieves a property by named `y` associated with a symbol table entry named `x` previously stored by the `putprop` statement.

`length(x)`

Returns the length of a list or a string.

`isarray(x)`

Returns non-zero if its parameter is an array, zero otherwise.

`isnumber(x)`

Returns non-zero if its parameter is a number, zero otherwise.

`isstring(x)`

Returns non-zero if its parameter is a string, zero otherwise.

`regex(x,re)`

The first parameter is a string, the second is a regular expression as recognized by the C `regex` function, also in the form of a string. It returns an array of beginning and length markers, corresponding to the positions in the first string that the regular expression matched.

`strip(x)`

Strips quotes from a string.

`substr(x,start,length)`

Returns a substring of its first parameter, starting at index `start` and up to `length` characters long, or the end of the string, whichever comes first.

`sprintf(format,args,...)`

Similar to the C `sprintf` function, this function formats its arguments into a string, which is returned.

Below is an example of the NMSL Extension Language input for the NMSL Compiler implementation, discussed in Chapter 7. This is only a partial listing of the Extension Language input, the full listing is over 1000 lines long.

```
declhead ( 1 )
    generic {
        error("Unknown declaration \"%s\" at line %d\n",
              $decl[1][1], $line);
    }
end declhead.

clause ((($1[1] == "requests") || ($1[1] == "modifies") ||
          ($1[1] == "traps")) &&
         ($clause[1] == "queries"))

    generic {
        if(length($*) != 1)
            error("Invalid \"%s\" clause at line %d\n",
                  $1[1], $line);
        let i = 2;
        let d = 1;
        do {
            if($1[i] == ",")
                error("extra comma at line %d\n", $line);
            let reqval[d] = $1[i];
            let d = d + 1;
            let i = i + 1;
            if((i < length($1)) && ($1[i] != ","))
                error("missing comma at line %d\n", $line);
            let i = i + 1;
            if(i > length($1))
                break;
        }
        return ($1[1], reqval);
    }
end clause.

clause (($1[1] == "process") &&
         (($decl[1][1] == "system") ||
          ($decl[1][1] == "domain")))
    generic {
        if(length($*) != 1)
            error("Invalid \"process\" clause at line %d\n",
                  $line);
        let curprop = getprop("__tmpprops__", $1[1]);
        let curprop[length(curprop)+1] = strip($1[2]);
        putprop("__tmpprops__", $1[1], curprop);
        if(length($1) == 2)
            return $1;
        if((length($1) == 5) &&
           ($1[3] == "(") && ($1[5] == ")")) {
            return($1[1], $1[2], $1[4]);
        }
        error("Invalid \"process\" clause at line %d\n", $line);
    }
    consistency {
        let dst = strip($decl[1][2]);
        let dstid = sprintf("['%s',%s]", dst, $2);
        printf("instanD('%s',process('%s'),%s).\n", dst, $2, dstid);
        /* check if this process had parameterized queries */
        let procinfo = get($2);
        if(procinfo != $null) {
            foreach x in procinfo[2] do {
                let proc = x[2];

                /* already generated code for the constant case */
                if(proc =~ "^\".*")
                    break;

                if(x[7] == $null)
                    let dsthost = "S1";
                else
                    let dsthost = sprintf("'%s'", x[7]);
                foreach y in x[5] do {
                    let freq = y[2];
                    printf("ref_eq(%s,Y1,readOnly,", dstid);
                    if(isnumber(freq))
                        printf("%d", freq);
                    else
                        printf("%s", freq);
                    printf(") :-\n");
                    let i = 1;
                    do {
                        if(i > length(procinfo[1]))
                            error("Help! internal queries error\n");
                        if(procinfo[1][i] == proc)
                            break;
                    }
                    if(i > length($3))
                        error(
                "Missing parameter to process \"%s\" at line %d\n",
                              $2, $line);
                    if($3[i] == "*")
                        let pname = "P";
                    else
                        let pname = sprintf("'%s'", $3[i]);
                    printf("\tinstanD(%s,process(%s),P1),\n",
                          dsthost, pname);
                    printf("\t(\n");
                    foreach r in x[3] do
                        printf("\t instan(%s,[%s],Y1);\n",
                              dsthost, r);
                    printf("\t instan(%s,[%s],Y1)).\n",
                          dsthost, x[4]);
                }
            }
        }
    }
end clause.
```

REFERENCES

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

[2] D. B. Berry, "Towards a Formal Basis for the Formal Development Method and the Ina Jo Specification Language," *IEEE Transactions on Software engineering* SE-**13**(2) pp. 184-201 (February 1987).

[3] 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).

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

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

[6] C. E. Catlett, "The NSFNET: Beginnings of a National Research Internet," *Academic Computing* 3(5) pp. 18-21 (January 1989).

[7] L. J. Cole, "Network Management as Described in Systems Network Architecture," *IEEE Infocom 86,* pp. 364-376 Miami, FL, (April 1986).

[8] S.W. Dietrich and D.S. Warren, "Extension Tables: Memo Relations in Logic Programming," *Proceedings of the Symposium on Logic Programming,* pp. 264-272 (1987).

[9] D. L. Estrin, "Inter-Organizational Networks: Stringing Wires Across Administrative Boundaries," *Computer Networks and ISDN Systems* 9 pp. 281-295 (January 1985).

[10] M. S. Feather, "Language Support for the Specification and Development of Composite Systems," *ACM Transactions on Programming Languages and Systems* 9(2) pp. 198-243 (April 1987).

[11] F. Fluckiger and C. Piney, "Principles of Control in a Distributed Network," *Networks 80, Online,* pp. 159-171 London, England, (June 1980).

[12] "Functional Specification and Description Language (SDL)," Blue book Fascicle X.1, Recommendation Z.100 and Annexes A,B,C and E, Recommendation Z.110, CCITT (1989).

[13] C.A.R. Hoare, "Communicating Sequential Processes," *Communications of the ACM* **21**(8) pp. 666-677 (August 1978).

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

[15] Information Processing Systems – Open Systems Interconnection, ''ESTELLE – A formal description technique based on an extended state transition model,'' ISO 9074, International Organization for Standardization (1989).

[16] Information Processing Systems – Open Systems Interconnection, ''LOTOS (Formal description technique based on the temporal ordering of obervational behavior),'' ISO 8807, International Organization for Standardization (August 1987).

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

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

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

[20] W. Joy, *An Introduction to the C shell,* 4.3BSD UNIX Programming Manual, University of California, Berkeley (1986).

[21] B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice Hall, Englewood Cliffs, NJ (1988).

[22] M. Kind, *StarKeeper Network management System – Overview,* AT&T Bell Labs (January 1987). Internal Memorandum.

[23] S. M. Klerer, ''The OSI Management Architecture: an Overview,'' *IEEE Network* 2(2) pp. 20-29 (March 1988).

[24] A. J. Kouyzer and A. K. van den Boogaart, ''The LOTOS framework for OSI Systems Management,'' *IFIP Integrated Network Management, II,* pp. 147-156 Washington, DC, (April 1991).

[25] S. T. Leutenegger, IBM, Private correspondence.

[26] D. C. Luckham and F. W. Henke, ''An Overview of Anna, a Specification Language for Ada,'' *IEEE Software* 2(2) pp. 99-22 (March 1985).

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

[28] K. McCloghrie and M. Rose, ''Structure and Identification of Management Information for TCP/IP-based Internets,'' RFC 1155, IETF Network Working Group (May 1990).

[29] R. E. Moore, ''Problem Detection, Isolation, and Notification in Systems Network Architecture,'' *IEEE Infocom 86*, pp. 377-381 Miami, FL, (April 1986).

[30] ''Network Management,'' *IBM Systems Journal* 27(1) pp. 1-85 (1988).

[31] M. T. Rose, *4BSD/ISODE SNMP Roadmap*, Performance Systems International, Inc.

[32] M. T. Rose, *The ISO Development Environment: User's Manual*, Performance Systems International, Inc.

[33] H. Tam, M. L. Schoffstall, W. Yeong, and M. S. Fedor, *Network Management Station Band Agent Implementation*, NYSERNet, Inc., Troy, NY (1989).

[34] R. B. Terwilliger and R. H. Campbell, ''PLEASE: Predicate Logic based ExecutAble SpEcifications,'' *Proceedings of the 1986 ACM Computer Science Conference*, pp. 349-358 Cincinnati, OH, (February 1986).

[35] U.S. Warrier, L. Besaw, L. LeBarre, and B.D. Handspicker, ''Common Management Information Services and Protocols for the Internet (CMOT) and (CMIP),'' RFC 1189, IETF Network Working Group (October 1990).

[36] J. Westcott, J. Buress, and V. Begg, ''Automated Network Management,'' *IEEE Infocom '85*, pp. 43-50 Washington, DC, (March 1985).

[37] S. Wilbur, ''Local area network management for distributed applications,'' *Computer Communications* 9(2) pp. 100-104 (April 1986).

[38] J. M. Wing, ''Writing Larch Interface Language Specifications,'' *ACM Transactions on Programming Languages and Systems* 9(1) pp. 1-24 (January 1987).

[39] P. Zave, ''An Operational Approach to Requirements Specification for Embedded Systems,'' *IEEE Transactions on Software Engineering* 8(3) pp. 250-269 (May 1982).