Specification and Verification of Network Managers for Large Internets

David L. Cohrs
Barton P. Miller

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

ABSTRACT

Large internet environments are increasing the difficulty of network management. Integrating increasing numbers of autonomous subnetworks (each with an increasing number of hosts) makes it more difficult to determine if the network managers of the subnetworks will interoperate correctly. We propose a high level, formal specification language, NMSL, as an aid in solving this problem. NMSL has two aspects of operation, a descriptive aspect and a prescriptive aspect. In its descriptive aspect, NMSL specifies abstractions of the network components and their instantiations, and verifies the consistency of such a specification. The abstractions include the data objects and processes in a network management system. These abstractions are instantiated on network elements. Network elements are grouped together in the specification of domains of administration. An extension mechanism is provided to allow for the specification of new management characteristics that the basic language cannot express. In its prescriptive aspect, NMSL generates configuration information directly from a consistent specification. This information is used to configure network management processes to make their operation consistent with their specifications. Standard management protocols (such as the emerging ISO or IETF standards) can be used to incorporate the configuration information into running management processes.

Research supported in part by an AT&T Ph.D Scholarship, National Science Foundation grants CCR-8703373 and CCR-8815928, Office of Naval Research grant N00014-89-J-1222, and a Digital Equipment Corporation External Research Grant.

1. INTRODUCTION

Computer networks are becoming more complex, connecting an ever increasing number of computers. As the size of computer networks grows, the need for network management tools to aid human administrators also grows. Current computer networks, while physically interconnected, are not joined together into one, homogeneous internet. In many cases, the protocols used in modern networks interoperate, but the networks themselves are actually divided into numerous, autonomous, independently administrated internets. We call these autonomous subnetworks *administrative domains*. Because of these different administrative domains, changes in the network may not be coordinated. There are no controls on the administrators of these domains to prevent them from causing global problems throughout the internet by making their own local management decisions. This autonomy of network administrative control is often necessary for pursuing independent research and other networking goals. Therefore, a computer network management system must be able to deal with the problems caused by this autonomy, and provide solutions that will not restrict autonomy, but will help make these autonomous networks coexist in a single, connected internet.

Our system, NMSL, addresses the problems caused by the large size and autonomy of current and future internets. NMSL includes both a method for specifying the relationships between different administrative domains and a method for verifying that these specifications are actually being adhered to in the network. We address both of these goals through the use of a high level specification language. This language can be used in two important ways. The *descriptive* aspect of the language allows the network administrators to describe the properties of the interfaces and configuration of the network management systems. These specifications can be formally verified for consistency to determine if the network administrators have configured their management software in a consistent manner. A specification is considered consistent if, for every data reference in the specification, there is a corresponding permission. Resource and timing requirements are included in the specification of references and permissions.

There is also a *prescriptive*, executable, aspect to the language. The specification can be translated into control commands that configure the various network management systems to provide the access permissions and frequencies necessary to achieve the global management goals. These control commands can be based on the emerging network management standards.

Our aim is to design and implement a tool that will solve the problems introduced by autonomy and multiple administrative domains. The tool we provide should handle the inherent distribution of control in the internet and must deal with the mutual distrust present in the internet. It should be able to scale to handle very large networks, on the order of 100,000 networks (and gateways), 100,000 to a million hosts, and 10,000 administrative domains. The autonomy of the domains of administration should be preserved. The tool should present the administrators with high level abstractions to specify the administrative relationships between the domains. It should also verify that the various rules and requirements given by the administrator are being met. Finally, the tool should map its abstractions onto the emerging standards, such as the OSI organizational model[11], or the IETF management framework[5]. In this way, it extends the standards' functionality by providing a high level, global management tool, using the protocols defined in the standard to perform the actual management of the network.

We are not directly addressing the important issues of security, or authentication. We also do not attempt to define the types of information the network management data should include, or the management protocols themselves.

This paper proceeds as follows. Section 2 describes related work in network management systems and standards, as well as work from the specification language literature. Section 3 describes the requirements for our specification language and gives an overview of the operation of the system. Section 4 discusses the descriptive aspects of the language, its grammer and semantics, how it is evaluated, and how we show the consistency of a management description. Section 5 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. Section 6 describes NMSL compiler operation, including code generation, and the NMSL language extension mechanism. Section 7 concludes with the project status, future directions of this research and our eventual goals.

2. RELATED WORK

Our research is based on ideas from 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 to specify many areas of computer development, for specifying both hardware and software designs.

2.1. NETWORK MANAGEMENT

Most previous computer network management systems concentrate on low level management issues, such as local area network management[8,19] and network address assignment[12], debugging local area communications software[3], and wide area network management[6,15,18]. Higher level issues, such as coordinating the interactions between network managers, and coordinating access controls, have been left for later work. Recently, researchers have begun to address the issue of providing higher level access to the low level information in the network management system, such as the work by Warrier[17], which provides a network management protocol independent query language. We feel that high level tools must be, as much as possible, independent of the low level protocols used to implement the network management.

Current research efforts are investigating the management needs of today's computer networks, and standards are emerging for the low level monitoring and control functions in the network. Some of these research projects also partially address the issue of administrative domains. Among these projects are network management systems for the TCP/IP Internet, the emerging OSI networks, and IBM's SNA networking product. The Internet and OSI network management systems are meant for use in large, distributed internets, with no central administration. IBM's network management product primarily addresses the needs of a single customer organization, and does not include general support on the problems of administrative domains. It does 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 DARPA Internet. As the name states, the protocol is simple, defining only a small number of messages. It includes a protection mechanism that allows flexibility in determining the accesses a remote domain of administration can make on data from a network element's database.

The OSI network management architecture[11], while still under development, also includes support for administrative domains in its current definition. Their organizational model assumes that management of the network will be distributed across different domains of administration. Each management domain can communicate with other domains via *ports*. Management domains may 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.

NetView[1], the SNA network management product from IBM, is a fully developed management product. 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.

These systems give us a good idea of the current directions in network management. The problem of domains of administration is addressed in each of them, but none of these systems include a mechanism for coordinating the configuration of the various network managers. 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. Our interest in specification languages is in using them 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 specification languages are Ina Jo[4], PAISLey[21], and Larch[20]. Various features of these languages are important to designing our network management specification language. Three other recent specification languages, Gist[7], PLEASE[16], Anna[13] and LOTOS[9] also have features in common with our work.

The specification languages mentioned are used primarily for specifying programming languages. Ina Jo, PAISLey and LOTOS use a *constructive* specification method. A constructive specification is one in which an abstract version of the algorithm used in the program is specified. PAISLey and Ina Jo can specify the interactions between multi-process programs, and include the ability to specify timing characteristics. LOTOS is used to specify OSI protocols, and also includes the ability to specify timing characteristics. Characteristics such as timing and other resource requirements and limits are important to any protocol specification, including network management.

Another specification approach is the *axiomatic* specification method. An axiomatic specification describes the properties a correct implementation must exhibit, without giving an algorithm for achieving those properties. The axiomatic method is more appropriate to our problem, because in specifying the configuration of a network management system, we are not concerned with the implementation of the system, but the properties an implementation must exhibit. Gist and Larch are two examples of axiomatic specification languages. Gist is also used to specify the interactions and operation a system of interacting components. It provides a definition of the desired properties of the program in an axiomatic way, rather than directly describing the construction of a program to achieve the goal of the specification. This approach is appropriate for specifying network management systems, and we use it as well. Larch takes the approach of giving a description, and, in addition, provides a two-level model for specifying programs. The lower level describes the underlying data abstractions, the upper, or interface, level describes how state changes occur. Larch also allows for specifying interactions between interacting, independent processes in its interface language specification. The two-level model gives Larch great generality, and separating the specification of abstractions from state or interaction specifications is useful, even in our specialized area. Neither of these language include the specification of properties such as physical resource requirements.

PLEASE, Anna and LOTOS include mechanisms to use the specifications to prototype or test the programs that they are specifying. This kind of specification language is called an *executable* specification language. PLEASE allows the programmer to convert the specification into a prototype. The user can then verify the specification by testing its prototype against a battery of test data. Compilers exist for LOTOS which can generate a protocol engine directly from the formal, LOTOS specification. For our application of specifying network management configurations, we plan to generate a configuration directly from its formal specification, and then install it in the network management system.

These formal specification languages give us an idea of the current technological level for specifying programs. While we are not using any of these languages directly, because specifying networks and network management is different from the ways one would specify an algebraic program, we are incorporating specific features into NMSL. Mechanisms for specifying process interaction are included some of the specification languages. The ability to specify timing constraints, as in PAISLey and LOTOS, is an important consideration. The axiomatic method, as well as separating the data abstractions from interactions, as in Larch, allows a cleaner, more easily understood specification. Executable specifications, as in PLEASE, can be used to generate code directly from a formal specification. These characteristics, in particular, are important to NMSL.

3. NMSL REQUIREMENTS AND OPERATION

NMSL is designed to meet the high level management needs of very large internets with a high level of autonomy. The requirements of such a system are discussed in this section. This is followed by an overview of NMSL's operation and a description of how the parts of NMSL will meet these requirements.

3.1. LANGUAGE REQUIREMENTS

NMSL's main goal is to detect inconsistencies in the configuration of the network and the network managers. As such, NMSL specifications are formal and axiomatic, allowing the use of automated proof techniques to find inconsistencies. The use of a axiomatic specification (as opposed to constructive) allows us to specify the interactions (often called *queries*) between network managers 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.

NMSL needs to be general and extensible. It must be able to represent the operation of a system of network managers, including the data objects (collectively referred to as the Management Information Base, or *MIB*), abstractions and encapsulations present in the management models. It must represent the instantiations of these data objects and types, and the interactions between the managers. Specifying interactions includes specifications of timing characteristics, the frequency of requests, bandwidth requirements, and the speed at which servers can process requests. NMSL should also be able to specify recursive queries (meaning that one server queries another server to process the query). Above all, NMSL must be easy to evaluate, to allow quick answers to questions of consistency and to scale to support the large networks of the future.

Providing general support for network management systems also requires support for more complex queries than outlined above. Some network management systems[11, 14], allow the network manager for a given network element to exist on some other network element. This type of network management is called *proxy* network management. Proxies are necessary because some network elements cannot respond to management queries directly. Such network elements include LAN bridges that do not support high level management protocols. Also included are protected (secure) systems that might not trust a foreign network manager. Specifying proxies requires NMSL to model the interactions between the proxy and the managed network element, as well as any data transformations made between the proxy protocol and the normal protocol. Once again, the specification of interactions must include the frequency of interaction and the use and availability of resources.

Making NMSL extensible means that the syntax and semantics of the language must be able to change to allow specification of new management information or protocols. An extension mechanism must allow for the introduction of new data or interaction abstractions. It must also define how the extension is to be evaluated, and must include rules to relate the extended specification to the specification information present in the basic language. The ways in which the new information affects the consistency definition must also be given. Finally, the prescriptive output of the extension must be defined.

Using a formal specification to prescribe the operation of the network managers at runtime makes NMSL even more useful. To be used in this way, the prescriptive output of NMSL must be as general as possible, being able to generate configuration data for various network managers. The NMSL compiler must include descriptions of how the formal specification can generate configuration data. To support extensions, any extension must also include a description of how that extension affects the prescriptive output.

In summary, NMSL is designed with two main aspects, a descriptive aspect, where it specifies network management configurations and checks the consistency of these configurations, and a prescriptive aspect, where it generates configuration information for the network managers. A formal, axiomatic language is required for the descriptive aspect, and must include specifications of the object types, and how these objects are instantiated. Extensions must be supported, via an extension language, to allow for the changing requirements of network managers. The prescriptive aspect requires NMSL to generate many different output configurations, depending on the type of network manager to be configured.

3.2. NMSL OPERATION OVERVIEW

Figure 3.1 gives a box diagram relating the parts of the NMSL system. The two aspects of NMSL's operation are divided roughly along a diagonal through the figure. The descriptive aspect includes the NMSL Compiler and the Consistency Checker. The prescriptive aspect includes the Configuration Generators. The boxes in the figure show the input and output of the NMSL system.

The Extension Language Specifications allow new specification types to be added to the basic NMSL repertoire. The system manager writes new specification types, relates them to the basic specifications, and describes the different compiler output in the Extension Language. This meets our goal of extensibility.

The compiler is central to the first aspect of NMSL operation. It takes as input the format of the basic and extended language specifications, along with the specifications for each part of the internet. The output of the compiler varies depending on the desires of the system administrator. There are two basic types of output, consistency facts and rules, and configuration information.

The consistency checker takes the facts and rules from the NMSL compiler, 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 manager. Determining consistency is a complicated operation, and the details of this operation are not part of this paper, although the high level operation of the consistency checker is discussed in Section 4.2.

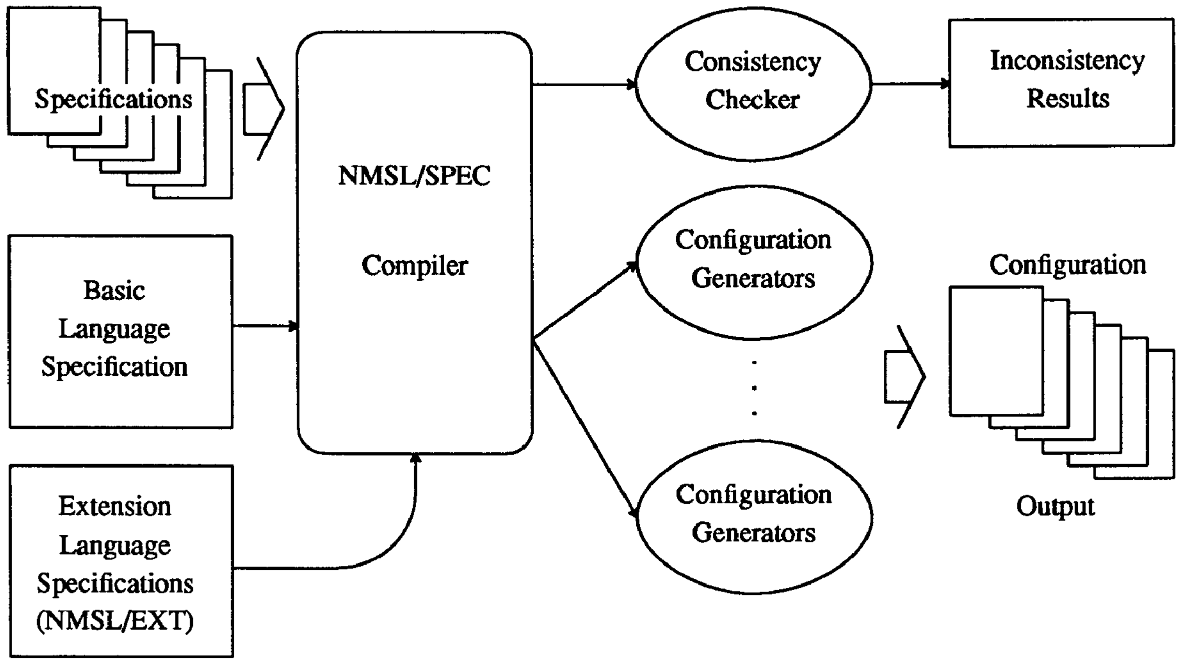

FIGURE 3.1. THE NMSL SYSTEM DESIGN

The configuration generators take the configuration output from the NMSL compiler and produce configuration information for the network managers. The configuration information is the prescriptive output of the NMSL system. This information can be shipped to the individual network managers, which can then be configured so as to meet the consistency requirements.

4. DESCRIPTIVE ASPECTS

This section explains the descriptive aspects of NMSL. The section is divided into two parts. The first part describes the basic language, its syntax and semantics, the abstractions it provides, and motivation for these basic facilities. The second part describes the evaluation mechanism and how it is used to determine the consistency of a NMSL specification.

4.1. THE BASIC LANGUAGE

The purpose of a NMSL specification is to denote the expected interactions that take place in a system of network managers. These specifications are written in an axiomatic way, in which the properties of the interactions are specified, not an algorithm used to perform the interactions. 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 lookup operation, it could modify some 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 and their expected behavior, where they are physically located in the internet, and the administrative properties and groupings of the parties involved.

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 a query. A data type specification must include the abstractions present in the database: how the data is grouped into complex data types, and the various representations that a data object can have. We call these *type specifications.*

We also need to specify the clients and servers in the system, the parties involved in queries. We call these *process specifications.* A process specification defines the queries that a process initiates, the relative order in which the queries are made, the frequency of queries, the parameters to the process and its queries. If the process is a server, the specification includes configuration information: which clients are allowed to query this server, how often, and for what data. A process specification may also include statements defining how instances of a data type specifications are transformed between their representations. A process, in our sense, is an abstraction similar to a type specification that must be instantiated at some place in the network.

The physical parts of the network are specified in *network element specifications.* A network element is any piece of hardware that can be connected to a network. A network element specification defines which processes are instantiated on that network element. 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 specifications*. A domain specification groups processes and network elements along administrative boundaries. Domains can overlap and nest. A domain specification 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 specifications is deliberate. Every network administrator has their own idea of how their 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, we need to specify how each part interacts, which causes the redundancy.

The separation of the specifications into abstractions (the data and process specifications) and instantiations (the network element and domain specifications) is also deliberate. This 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 pointed out in Larch[20], 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.

4.1.1. NMSL SYNTAX

The syntax of the NMSL specifications 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. The syntax of the statements is presented as BNF-style grammar rules. In the examples, SNMP and the IETF MIB are used exclusively. This is only for simplicity; NMSL works equally well with the OSI management framework.

4.1.2. TYPE SPECIFICATIONS

The syntax of NMSL type specifications is based on the ISO Abstract Syntax Notation One (ASN.1)[10]. ASN.1 has the advantages that it is general, machine architecture independent, and well known. It is used to specify the variables 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 use with NMSL. ASN.1 macro descriptions are not supported, since the NMSL extension mechanism fulfills this role.

```
TypeSpec    ::= "type" TypeName "::="
                Asn1-body ";"
                AccessSpec
                "end" "type" TypeName "."
TypeName    ::= String
AccessSpec  ::= "access" AType ";" | EMPTY
AType       ::= "Any" | "ReadOnly" |
                "WriteOnly" | "None"
```

FIGURE 4.1. FORMAT OF TYPE SPECIFICATIONS

A NMSL type specification is formed by the keyword `type` followed by a standard ASN.1 type description followed by a trailer, as shown in Figure 4.1. The keyword, `type`, starts a type specification. The `TypeName` token is any valid ASN.1 type identifier, and must be the same in both positions in the type specification. The ASN.1 specification given in `Asn1-body` provides the data grouping and representation specifications. Its format is described in the ASN.1 standard. `AccessSpec` allows simple access checks to be made when a request in a process specification is made to access data. This clause can be modified via the extension language to handle more complex access requirements. If no `AccessSpec` is present, the access characteristics are inherited from a containing type, as is shown in the example, below.

As an example of the use of a type specification, consider the specification of the IP Address Table that defines the IP Addresses for a given entity. The example in Figure 4.2 is derived from the TCP/IP MIB[14]. 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.

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 is inherited; the access mode of `IpAddrEntry` was not specified, but this was used in the definition of an `ipAddrTable`, so

```
type ipAddrTable ::=
    SEQUENCE of IpAddrEntry;
    access ReadOnly;
end type ipAddrTable.

type IpAddrEntry ::=
    SEQUENCE {
        ipAdEntAddr         IpAddress,
        ipAdEntIfIndex      INTEGER,
        ipAdEntNetMask      IpAddress,
        ipAdEntBcastAddr    INTEGER
    };
end type IpAddrEntry.
```

FIGURE 4.2. EXAMPLE TYPE SPECIFICATION

all IpAddrEntry elements in an ipAddrTable are read-only. The integration of the type specifications with the other specifications is given in the following sections.

4.1.3. PROCESS SPECIFICATIONS

A process specification 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 specification defines a new abstraction in the same way a data type specification defines a new abstraction. Processes can be instantiated in a network element or domain specification.

A portion of the grammar for the process specifications is shown in Figure 4.3. A network management process can perform many functions, but we are only interested in its interactions with other processes and the network management data to which it has access. The interactions are specified using the QrySpecs clause. Figure 4.3 only shows queries that retrieve data; the full language supports queries that modify data and those that cause remote execution as well. A QrySpec specifies the parameters to the query (the using clause), the expected response (the requests clause) and the timing characteristics of the query (the frequency clause).

```
ProcessSpec::= "process" ProcName
               ParamSpec "::="
               PSpecs
               "end" "process" ProcName "."
ParamSpec   ::= "(" ParamList ")" | EMPTY
ParamList   ::= ParamList "," Param | Param
PSpecs      ::= ViewSpec | ExSpecs | QrySpecs
Param       ::= NMSLVariable ":" TypeName
ViewSpec    ::= "supports" VList ";" | EMPTY
ExSpecs     ::= ExSpecs ExSpec | EMPTY
ExSpec      ::= "exports" VList
                "to" DomainName
                "access" AccessType
                "frequency" Freq ";"
QrySpecs    ::= QrySpecs QrySpec | EMPTY
QrySpec     ::= "queries" ProcName
                "requests" VList
                "using" AsgnVList
                "frequency" Freq ";"
VList       ::= VList "," VarID | VarID
Freq        ::= BoundSpec Float TimeSpec
              |  "infrequent"
BoundSpec   ::= "<" | "<=" | ">" | ">=" | EMPTY
TimeSpec    ::= "hours" | "minutes" |
                "seconds"
```

FIGURE 4.3. FORMAT OF PROCESS SPECIFICATIONS

Some processes store, have direct control over, and can answer queries for management data. We say that such processes *support* a part of the MIB. The process specification includes clauses to specify this supported management data as well. The supported MIB variables are listed in a ViewSpec. The access permissions other processes have with respect to this supported data are specified with an ExportSpec. The ExportSpec makes use of the administrative groupings specified in a domain specification.

Figure 4.4 shows a simple example of a SNMP agent and an application process.[1] The agent process, snmpdReadOnly, is a simple SNMP agent that exports all of its variables read-only to the "public" 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 once every 5 minutes.

The application process, snmpaddr, retrieves information about a single network interface connected to a network element. The application has two parameters, SysAddr and Dest, which are assigned values when the application is instantiated (executed). Snmpaddr sends a query to an agent, identified by the SysAddr variable. The query requests the contents of the entire IpAddrEntry sequence with the selection criterion that the sequence returned have ipAdEntAddr equal to the value of Dest (specified in the using clause). This application is expected to make queries infrequently. Other important information needed during the query, such as the domain in which the process executes, is specified in a network element or domain specification when the process is instantiated.

```
process snmpdReadOnly ::=
  supports mgmt.mib; -- entire MIB subtree
  exports mgmt.mib to "public"
    access ReadOnly
    frequency >= 5 minutes;
end process snmpdReadOnly.
process snmpaddr(
    SysAddr: Process; Dest: IpAddress) ::=
  queries SysAddr
    requests
      mgmt.mib.ip.ipAddrTable.IpAddrEntry
    using
      mgmt.mib.ip.ipAddrTable.
      IpAddrEntry.ipAdEntAddr := Dest
    frequency infrequent;
end process snmpaddr.
```

FIGURE 4.4. EXAMPLE PROCESS SPECIFICATIONS

[1]An agent process stores management data and responds to requests while an application process is one that initiates requests but does not store management data itself.

4.1.4. NETWORK ELEMENT SPECIFICATIONS

Network element specifications describe the physical properties of the network. Each specification shows the details for a single computer (or terminal, printer, router or other device that could be connected to a network) and its connections to the 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.

A portion of the grammar for a network element specification is shown in Figure 4.5. The `SystemSpec` groups the subspecifications into a network element specification. The properties of the hardware are given in the `HardConf`, including the cpu type and a list of network interfaces. Each `IfSpec` lists one network interface, an identifier, the physical network to which the interface connects, the type of interface, and its 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 `SoftConf` gives a description of the operating system for this network element. The `ViewSpec` lists the portion of the MIB that this network element supports. Each network management process is that is instantiated on this network element is listed in the `ProcSpecs`.

A simple example of a network element specification is shown in Figure 4.6. This network element, called `romano.cs.wisc.edu`, has single interface, connecting it to a 10Mbps ethernet named `wisc-research`. It runs the SunOS 4.0.1 operating system. It supports most of the IETF MIB, but does not support the EGP variables. `Romano.cs.wisc.edu` runs the read-only SNMP agent process used in Figure 4.4.

```
SystemSpec ::= "system" SystemID "::="
               HardConf SoftConf ViewSpec
               ProcSpecs
               "end" "system" SystemID "."
HardConf   ::= "cpu" String ";" IfSpecs
IfSpecs    ::= IfSpecs IfSpec | EMPTY
IfSpec     ::= "interface" String
               "net" String
               "protocols" PrPList
               "type" String
               "speed" Integer "bps" ";"
SoftConf   ::= "opsys" String
               "version" String ";"
ProcSpecs  ::= ProcSpecs ProcSpec | EMPTY
ProcSpec   ::= "process" ProcInvoke ";"
ProcInvoke ::= ProcName ProcParams
ProcParams ::= "(" PrPList ")" | EMPTY
PrPList    ::= PrPList "," PrParam | PrParam
PrParam    ::= String | Integer | Float
```

FIGURE 4.5. FORMAT OF NETWORK ELEMENT SPECIFICATIONS

4.1.5. DOMAIN SPECIFICATIONS

Domain specifications describe the administrative groupings found in networks. Domains may contain other domains (sub-domains), network elements and processes. Processes can be specified in the domain specification (as opposed to the network element specification) to allow them to be instantiated in the domain without specifying which network element is being used. A domain specification also lists how its members relate administratively to other domains.

A portion of the grammar for a domain specification is shown in Figure 4.7. The members of a domain are given in the `MembSpecs` production. Subdomains, like other members, are specified separately and are only referenced in the domain specification. The `ExSpecs` production specifies which parts of the MIB that are available in this domain can be accessed by processes in other domains. The `ExSpecs` production is redundant; is also given in an process specification. It is given here as part of the consistency mechanism. It can also further restrict how other domains may access the members of this one.

```
system "romano.cs.wisc.edu" ::=
    cpu sparc;
    interface ie0 net wisc-research
        type ethernet-csmacd
        speed 10000000 bps;
    opsys SunOS version 4.0.1;
    supports
        mgmt.mib.system, mgmt.mib.at,
        mgmt.mib.interfaces,
        mgmt.mib.ip, mgmt.mib.icmp,
        mgmt.mib.tcp, mgmt.mib.udp;
    process snmpdReadOnly;
end system "romano.cs.wisc.edu".
```

FIGURE 4.6. EXAMPLE NETWORK ELEMENT SPECIFICATION

```
DomainSpec ::= "domain" DomainName "::="
               MembSpecs
               ExSpecs
               "end" "domain" DomainName "."
MembSpecs  ::= MembSpecs MembSpec | EMPTY
MembSpec   ::= "process" ProcInvoke ";"
             |  "domain" DomainName ";"
             |  "system" SystemID ";"
```

FIGURE 4.7. FORMAT OF DOMAIN SPECIFICATIONS

```
domain wisc-cs ::=
    system romano.cs.wisc.edu;
    system cs.wisc.edu;
    process snmpaddr(*, *);
    exports mgmt.mib to "public"
        access ReadOnly
        frequency >= 5 minutes;
end domain wisc-cs.
```

FIGURE 4.8. EXAMPLE DOMAIN SPECIFICATION

A simple example of a domain specification is given in Figure 4.8. In this example, the domain, wisc-cs, contains two network elements. It also contains an instance of the SNMP application process, snmpaddr, specified in Figure 4.4. In this example, the value of the parameters to the snmpaddr process are not known at the time the specification is written. We use the special token, "*", to denote that the values will be set at the time that the process is run. 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 every 5 minutes.

4.2. EVALUATION

Given a network specification, the NMSL system must tell the system administrator if the specification is consistent. If it is not, the causes for inconsistency should be displayed. This is the task of the NMSL Consistency Checker. The Consistency Checker takes as input the consistency output from the NMSL Compiler. The consistency output is in the form of statements of a logic programming language. The Consistency Checker adds rules to these logic statements that describe the meaning of consistency. It then passes the rules to the logic interpreter, along with a logical statement requesting the interpreter to find inconsistencies in the specification.

The NMSL Consistency Checker is a front end for the Prolog dialect, CLP(**R**)[2]. CLP(**R**) was chosen because of its speed in performing logical deduction, and its ability to check numeric constraints over the real numbers. Numeric constraints are important for specifying timing and other resource limitations of interactions. The closed-world model used by CLP(**R**) is appropriate for a NMSL specification, because we are interested in determining if the parts of the specification are consistent with each other, not with all possible specifications.

The NMSL compiler converts the NMSL specification into CLP(**R**) statements. The output from the compiler is a collection of logical facts and rules. The Consistency Checker adds statements describing the consistency of any NMSL specification to this output and executes the CLP(**R**) interpreter. The interpreter performs the consistency check.

Consistency of a NMSL specification is defined in terms of the relationships between the types, processes, network elements and domains in the specification. There are six relationships in our consistency model: containment, instantiation, two reference relations and two permission relations. These relationships are shown, along with their definitions, in Figure 4.9. In the course of a proof, these are reduced to just ref_eq and perm_eq relationships. A NMSL specification is said to be *consistent* if, for every reference relationship, there is a corresponding permission.

Part of a reference or a permission relationship are resource limitations, such as time, frequency, size or bandwidth. The type of access is also specified. In their current form, references and permissions only include frequency and access type. Other resource requirements would appear as additional parameters to the relationships.

Three types of rules are employed during a consistency proof: transitivity, distribution, and reduction rules. The containment relationship is transitive. The distributive rules are used to distribute containment and instantiation over each other, and over reference and permission. The reduction rules relate reference and permission relationships.

The proof performed by the Consistency Checker is really a proof of inconsistency. When an inconsistency is proved, it is reported to the system administrator. At this point, the offending parts of the specification must be corrected. If no inconsistencies are found, the specification is consistent. This method of proving consistency is viable because of our closed-world assumption. At this time, we are formally defining the relationship between containment, reference and permission (that is, how references by the members of a domain are affected by a permission given to their containing domain). Once this is completed, we will show the completeness of our model for the class of network management systems represented by current standards. These results will be presented in future paper.

The NMSL Consistency Checker can also be used in a speculative role. Consider the scenario where a network administrator is about to connect a new organization to the internet. The load this organization will place on the internet is basically unknown. However, the administrator can make a specification of the new organization's expected interactions with the existing parts of the internet. If summary data is available for the existing internet, approximate values can be used to determine the amount

contains(X,Y)	X contains Y
instan(X,Y,Z)	X instantiates Y with unique ID Z
ref_eq(X,Y,A,T)	It is possible that X references Y for access A every T seconds
ref_gt(X,Y,A,T)	... at most every T seconds
perm_eq(X,Y,A,T)	X has permission to reference Y for access A every T seconds
perm_gt(X,Y,A,T)	... at most every T seconds

FIGURE 4.9. NMSL LOGICAL RELATIONSHIPS

of traffic generated, for example. This specification can be tested with the existing internet specifications by the NMSL Consistency Checker. It will be able to tell the administrator if there are inconsistencies in the new specification.

Another speculative use of the Consistency Checker utilizes CLP(**R**)'s ability to run the consistency check in reverse. Once again, this is used to check a new specification against an existing, consistent specification. In this case, we would make the consistency of the combined specification a premise of the proof, and ask CLP(**R**) to solve for the parameters to the references and permissions of the new specification that satisfy this premise. Using the Consistency Checker in a speculative role could help to ease the pain of a growing internet.

5. PRESCRIPTIVE ASPECTS

Once a specification is determined to be consistent, the specification can be executed to configure the network management processes. As in the case of consistency checking, a NMSL Configuration Generator takes output from the NMSL Compiler and uses it to configure a network manager. The compiler cannot configure a network manager directly, because this requires knowledge of data formats, protocols and authentication issues that the compiler should not have to know about. The NMSL Configuration Generator 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.

The operation of the prescriptive aspect is simple. The NMSL Compiler is rerun with a parameter requesting configuration output of a specific type. It parses the specification just as it would to generate consistency checking output. The output it generates, however, is configuration output instead of Prolog rules. The way in which these different types of output are specified is described in Section 6. A NMSL Configuration Generator takes the configuration output and transmits it to each appropriate network element or process. Ideally, this would be achieved by initiating a connection to a network management process on each affected network element, authenticating the Configuration Generator as a trusted process, and sending, via the normal network management protocol, the configuration information. 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.

In practice, it may be too time consuming to generate the configuration output from one central location. This depends on the frequency of changes to the management specification, and the number of network managers that need configuration. It may be possible to perform the configuration phase in a distributed manner. If a process's configuration depends only on its own specification, the configuration information for that process can be generated from its specification alone, and can be generated on the network element on which the process executes. In this way, the benefit of using the executable specification is still achieved, but the work of generating the configuration information does not swamp a single computer. At this point in our research, we do have not determined how we would distribute NMSL's prescriptive operation.

6. THE NMSL COMPILER OPERATION AND THE EXTENSION MECHANISM

The grammar recognized by the NMSL Compiler is less specific than the previous sections indicate. The examples in Section 4 show that each type of specification has the same structure. Each specification has the same format (a header, a body and a trailer); each clause of a specification is made up of subclauses that begin with a keyword and are followed by a list of parameters. The NMSL Compiler actually recognizes this less specific grammar. Parsing the specifications in this form has the advantage that it simplifies the generation of multiple types of output and simplifies the mechanism that adds extensions to the basic grammar.

6.1. SYNTACTIC AND SEMANTIC CHECKING

The NMSL Compiler operates in two passes. In the first pass, it parses the input specifications according to the generalized grammar shown in Figure 6.1. In the process of parsing the input, the compiler builds a parse tree for the specifications, but it does not attempt any semantic analysis of the specifications. This means that any group of tokens will be accepted by the parsing pass, provided that the group of tokens matches the basic format of the NMSL grammar. The task of differentiating between the specifications and clauses is left for the second pass.

```
decls        ::= decls decl  | EMPTY
decl         ::= decltype declname
                 declparams "::="
                 clauses
                 "end" decltype declname "."
declparams   ::= "(" list ")"  | EMPTY
clauses      ::= clauses clause  | EMPTY
clause       ::= subclauses ";"
subclauses   ::= subclauses subclause
             |    subclause
subclause    ::= string list
             |  asn1-id asn1-subclause
list         ::= list "," listelement
             |    listelement
listelement  ::= token special token | token
token        ::= string | number | special
```

FIGURE 6.1. GENERALIZED NMSL GRAMMAR

Associated with each production shown in Figure 6.1 is a list of *actions*. These actions are executed in the second pass of the compiler. When they are executed, they are passed the context in which the production was used and the parameters corresponding to a list of the tokens generated by the current production. The actions have two tasks. Their first task is to determine if the specifications parsed by the first pass are valid. The type of the declaration (`decltype` in Figure 6.1) must be valid, as must the name of the declaration, the parameters, and each of the clauses. The names of each type of declaration and clause can be checked by a simple table lookup. Determining if the contents of a clause is correct requires executing a clause-specific semantic check.

The actions that detect semantic errors are common to all runs of the compiler, and do not change depending on the type of output generated. We call them *generic* actions. The generic actions statements are *tagged* with the identifier, `generic`, in the compiler's tables. Generic actions can also perform bookkeeping operations, such as accessing the symbol table. Output (code generation) is performed by a collection of *output specific* actions.

6.2. CODE GENERATION

The output specific actions allow the second pass to perform its other task, generating output. In the compiler's tables, each output specific action is tagged with an identifier for the type of output that it generates. When the compiler is run, one of its parameters specifies the type of output desired. For example, requesting consistency output causes the actions tagged `consistency` to be executed, and Prolog rules to be generated. Other actions, such as an action tagged `BartsSnmpd` would be executed only if configuration output for Bart's SNMP daemon were being generated. Each run of the compiler executes the generic actions and one type of output specific action.

6.3. THE NMSL EXTENSION MECHANISM

The basic NMSL language allows the specification of a wide variety of different network management configurations. However, we cannot anticipate all possible configurations or forms of specification. It is also time consuming to enter groups of similar specifications; a macro facility would be helpful. The NMSL extension language addresses these two issues.

The extension input to the NMSL Compiler (see Figure 3.1) is a simple list of typed *keywords* and *actions*. This format was chosen because the format of basic grammar identifies declarations and clauses. Further semantic processing is based on the keyword identifying the declaration or clause. The compiler creates an internal, extended keyword table and an extended action table from the extension language input when it begins execution. The extended tables are used by the compiler in its second pass. When it performs an action for a production, the compiler consults the extended keyword table before it consults the basic list. If an entry in the extension matches, the compiler performs the action statement associated with the extended keyword (if one exists). This process corresponds to the way basic declarations and clauses are processed. If the keyword for an extension is different from all existing keywords, then the extension extends the language; if the keyword is the same as an existing keyword, it overrides the basic language. However, only the actions specified in the extension override the basic actions. For example, an extension that specifies the keyword `queries` (`queries` is a keyword in the basic language) and a single action tagged `DavesSnmpd` will not override the basic `generic` action for the `queries` clause, but it will override an existing action tagged `DavesSnmpd`. This allows new types of output for a production to be specified in an extension without changing that production's generic processing.

The simplified grammar can increase the complexity of the action statements. An action statement for a clause must determine if the clause is valid in the context in which it is used. Additionally, if an action overrides an action of a basic clause, it must include code to handle all contexts in which the clause can appear. Usually, the semantics of a clause are the same in each context, so the action statements do not become much more complex.

The NMSL extension language extends the declarations and clauses recognized by the NMSL Compiler. This is done by prepending the extension keyword and action tables to the basic tables. Prepending allows extensions to override the basic actions. Because the semantics and output actions, not grammar, are specified in the extension, we preserve the look and feel of the NMSL language. We feel this gives us the growth potential we need from the extension language, without allowing more flexibility in an extension than necessary.

7. CONCLUSIONS AND FUTURE WORK

NMSL attacks the problem of managing large internets by providing a high level, formal specification language. We have shown the two aspects of NMSL, the descriptive aspect, and the prescriptive aspect. In its descriptive aspect, we have shown that the formal, axiomatic language allows the specification of management abstractions and their instantiations in the internet. The abstractions include data objects and network management processes. NMSL supports the specification of network elements and administrative domains, and the instantiation of processes in these specifications. Most importantly, we showed how these specifications are checked for consistency with the NMSL Consistency Checker.

To address changing network environments and standards, NMSL includes an extension mechanism. We discussed the way this mechanism allows the specification of abstractions and objects that have not been anticipated in the basic language. Extensions can override clauses and declarations in the basic language in a controlled way.

We also showed how the look and feel of the basic language is preserved in an extension through the use of a general parser that parses a specification without considering its semantic content.

In NMSL's prescriptive aspect, a NMSL Configuration Generator generates configuration information directly from a consistent specification. We described the support for multiple types of action statements for each production in the NMSL grammar. These actions generate different types of output for a single specification, including many types of configuration information. Finally, we outlined the way this information can be sent directly to network management processes, increasing the probability that these processes will interoperate in the desired way.

Research into the NMSL system is still at its early stages. Our immediate task complete our formal definition of the meaning of a consistent NMSL specification. Various details of the clauses in the basic language must also be worked out. We plan to implement the system as it is described here, including the Compiler, the Consistency Checker, and at least one Configuration Generator. The implementation is planned for use with SNMP in a TCP/IP environment.

REFERENCES

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

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

[3] J.A. Barchanski, "Expert Systems for Local Computer Network Software Debugging," *Proceedings of the 1987 IEEE 12th Conference on Local Computer Networks*, pp. 154-159 Minneapolis, MN, (October 1987).

[4] 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).

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

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

[7] 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).

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

[9] 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).

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

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

[12] W.M. Louks, W.I. Kwak, and Z.G. Vranesic, "Implementation of a Dynamic Address Assignment Protocol in a Local Area Network," *Computer Networks and ISDN Systems* **11**(3) pp. 133-146 (July 1986).

[13] 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).

[14] K. McCloghrie and M. Rose, "Management Information Base for Network Management of TCP/IP-based Internets," RFC 1066, IETF Network Working Group (August 1988).

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

[16] 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).

[17] U. S. Warrier, P. A. Relan, O. Berry, and J. Bannister, "A Network Management Language for OSI Networks," *ACM SIGCOMM 88*, Stanford, CA, (August 1988).

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

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

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

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