Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

INTELLIGENT NETWORK MANAGEMENT: INFORMATION INFRASTRUCTURE AND MANAGEMENT APPLICATIONS

Y.-D. Lin
M. Gerla

July 1993
CSD-930024

Intelligent Network Management: Information Infrastructure and Management Applications

Ying-Dar Lin[1]
Mario Gerla
Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90024

Abstract

The key issues in network management are the representation and sharing of management information and the automatic management mechanisms based on the underlying information infrastructure. We propose the methodologies of (i) global view abstraction for the management information infrastructure and (ii) learning and inference for automatic and adaptive network management. Views are global management information constructed via logical rules. Management applications access these views to learn network patterns and reason on the discovered patterns and pre-specified domain knowledge to predict network status, disgnose problems, and trigger control actions. The proposed scheme is meant to operate on the OSI standard management architecture where management information is stored in object-oriented databases. Management knowledge base which includes network patterns, abstract view definition, and domain knowledge is represented as a set of logical rules. A network management system, GlobeView, and a case study on ATM network topology tuning are presented.

1 Introduction

Recent progress in network management is the recognition of the need to use standardized databases for storing network management information and a standardized protocol to access the stored information. This solves the interoperability problem [CDFS88,MR90,Ros90,ISO90A,ISO90B]. Definition and implementation of MIB (Management Information Base) and CMIP (Common Management Information Protocol) are on-going efforts [HBRD93,MBL93]. However, one problem

[1]The author is now with Dept. of Computer and Information Science, National Chiao-Tung University, Taiwan.

remains. Namely, how to abstract the *global* management information from the distributed management information infrastructure? The management system needs to provide the users, either human managers or management applications, the capability to *view* and *control* the entire network via a *single* query command.

On top this information platform, automatic management applications deal with specific management tasks. However, management tasks like performance tuning and fault diagnosis require good understanding of traffic patterns and knowledge of causality which we might not have models to describe. In general, a pattern can exist in client-server interactions, temporal and geographical traffic distribution, traffic/performance relationship, performance correlation between network entities, alarms or faults correlations, and some hidden causal relationships. In [LTHG93], we propose the HAP model for packet arrivals on a short time scale for real-time control. Here we are looking at a longer time scale and assuming that we do not have a model for the behaviors of traffic sources.

The importance of understanding, and furthermore capturing, patterns stems from several different reasons. Phenomena can be explained more precisely and problems can be diagnosed. Knowing the dynamics within the system will enable us to predict the system behavior and perform adaptive control. In an adjustable system, we can further tune the system according to the pattern if some status is foreseen to occur. However, to understand the network patterns, we need the historical information of the network. Three issues arise here. First is the representation issue: in what format are we going to store the current/historical information and the discovered patterns? Second is the learning or knowledge acquisition issue: how are we going to discover the patterns from the stored information trace? Third is the inference or knowledge use issue: based on the management information and the captured patterns, what kind of automatic control/management mechanisms can be built?

Given that we want to abstract global management information and construct automatic and adaptive management applications, we propose a framework for a network management system with learning and inference abilities, where learning is to capture network patterns and inference is to reason on the discovered patterns and pre-specified knowledge in order to access virtual global objects, predict network status, trigger control actions, and diagnose problems. The proposed scheme is meant to operate on the standard management architecture where management information is stored in object-oriented databases. Management knowledge base which includes network patterns, abstract view definition, and domain knowledge is represented as a set of logical rules. These rules are triggered by the facts in databases and queries from management applications. The goal is autonomous network management by expert systems with learning capability.

Section 2 highlights the network management issues and their recent progress. The induction/deduction approach is proposed in section 3. In section 4, network patterns are classified and the pattern discovery process is described. The backward deduction for diagnosis and ab-

straction, and the forward deduction for prediction and control are illustrated. The architectural aspects of the proposed scheme and its operation on the standard management model are described in section 5. The techniques to build management information infrastructure and management applications are detailed in section 6 and 7. Section 8 present an experiment of pattern discovery in LAN environment. The GlobeView implementation, listed in Appendix, and management application on topology tuning are detailed in section 9. Parts of the report are published in [GL91A,GL91B,LG92,LG93].

2 Problem Domain: Network Management

Unlike real-time control, management is not an essential component to simply make the system work. That is, a system can continue to function, at least for a period of time, without the management subsystem. However, what were once highly tuned systems may gradually degenerate to an inefficient state. Not only a software/hardware failure but also performance degradation can be a system problem. Thus, the task of the management subsystem is to keep track of the system status, which includes both configuration and performance, and trigger control actions when necessary. We can divide the management process into the monitoring process and the control process. The monitoring process involves collecting information about the system's short-term/long-term behavior and low-level/high-level status, filtering out unimportant information to reduce stored data volume, and interpreting the semantics of the collected information. The control process affects the state of the system according to the interpreted information to achieve a desired outcome. In the above processes, we find that there are two major issues in network management: management information infrastructure and automatic/adaptive management schemes.

A. Management Information Infrastructure

Any network management system must be constructed on top of the underlying management information model on which the representation schemes and operations are based. Given that a network is a distributed, and maybe heterogeneous, environment, several issues are confronted when designing the infrastructure of the network management information:

- **Management information representation**: In what form can the information be stored in network entities and exchanged between network entities? What kind of management information needs to be supported? Do the format and the content have to be standardized for information sharing?
- **Heterogeneity of protocol stacks**: How can machines with different protocols interoperate to share management information?

- **Information distribution strategy**: What is the mechanism for information sharing between network entities and the management system? Should the management system keep a global view of the network at all time or reconstruct it, when needed, from local views of network entities?

Here we are facing problems similar to information sharing problems in a traditional file system, with multiple applications, where an application must know the structure of the files it is operating with. If one particular application needs to modify the structure of a file, all the other applications using that file have to be changed. The solution to avoid this led to the evolution of database systems which contain the files, the file structures, and the primitives to access them. The separation between data and applications provides *data independence* for applications [Ull88]. By the same philosophy, data independence for network protocol stacks and management applications can be supported by the management information databases and their access protocol. The database primitives and the access protocol form the information access primitives for protocol stacks and management applications. As information may be shared by distributed heterogeneous network entities, management databases and access protocol must be standardized.

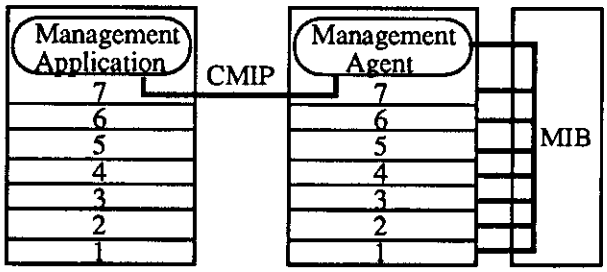


Figure 1: OSI Management Model

The open-networking community has settled on a management model that places a MIB on each network node and manages these MIB's remotely with application level protocols [ISO90A,ISO90B,CPW89]. The widely accepted OSI management model is illustrated in Figure 1. A MIB, an abstract image of the local management objects, is supported by each seven-layer OSI node. Objects are manipulated by the application-layer management protocol CMIP, which uses RPC (Remote Procedure Call) protocol. Changing the attribute values in a MIB will result in changing the status of the physical network entities. For example, setting the status attribute of link 537 to off can disable that link.

Because of the hierarchical nature of network entities and their sub-entities, both ISO and Internet models organize network management information into a hierarchical structure. ISO even encapsulates this hierarchical model into object-oriented databases in order to hide the heterogeneity

of network entities away from the protocol stacks and management applications. In object-oriented databases, the following concepts are supported: (i) subtype hierarchy (by record formation and set formation) and method inheritance, (ii) encapsulation, and (iii) object identity [Ull88]. An *object class* is associated with a set of *methods* operating on the *object instances* of this object class. An object subclass inherits the set of methods from its parent object class. The encapsulation of the heterogeneity of network entities is achieved by the sets of methods.

The adopted architecture solves the problems of **information representation** and **heterogeneity of protocol stacks**, but the problem of **information distribution strategy** remains. Given the standard platform, we still need a mechanism to construct the global views for the management applications. This is one of the problems we want to solve.

B. Automatic and Adaptive Management

Although the infrastructure of network management is agreed upon regarding the standard MIBs and CMIP, little was done to define how to use this platform in specific network management problems: performance, configuration, fault, accounting, security, etc. Several researchers have adopted expert systems with domain knowledge represented as a set of logical rules capturing network management model to cope with fault localization and correction [EEM89,Goy91,Lew93]. In these systems, network messages containing "trouble tickets" are sent to the expert system. This expert system then reasons on the trouble tickets and network configuration to find the possible fault locations and the recovery procedures. The effectiveness of these systems depends heavily on encoding the problem-solving knowledge in the network domain. The goal of these expert systems is an automatic fault management system to enhance or even replace human intervention.

Other network management problems also need automation. The maintenance of a large number of objects in MIBs needs to be done automatically to keep the status information up-to-date. Configuration management applications can then easily identify and update objects. This in turn changes the configuration of network entities. Either remedial or preventive performance management schemes need to be triggered automatically by performance alarms or traffic forecasting, which again depends on automatic interpretation of performance and traffic measurements. This measurement interpretation implies that the system needs to keep track of the *network patterns* and perform *adaptive control*. The ultimate goal for network management should be a self-managed and self-adjustable network.

3 Proposed Approach

Our approach to solve the above two network management problems: information distribution strategy and automatic/adaptive management is to incorporate learning and inference abilities

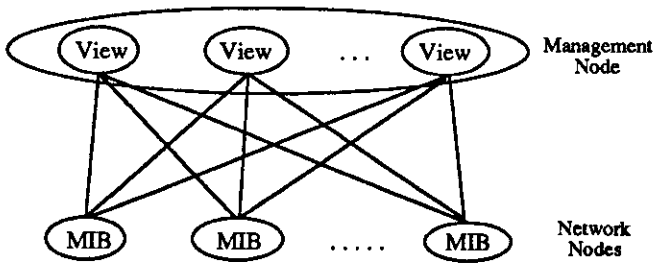


Figure 2: Information Infrastructure: Global Views and Local MIBs

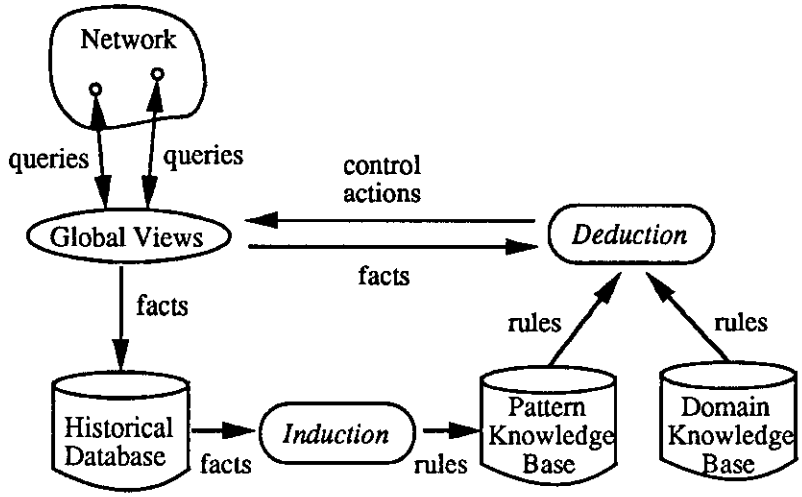


Figure 3: Induction/Deduction in Network Management

into network management systems to automate the process of global view construction, measurement interpretation, problem forecasting, problem diagnosis, and decision making. To build the information infrastructure, a set of global views is constructed. A *global view* is a virtual object class defined from all local MIBs via logical rules. These global views serve as *windows* through which management applications can access physical network entities. Figure 2 shows a set of global views constructed from local MIBs. To equip the system with automatic and adaptive abilities, network patterns are learned from a historical database which contains a chronological measurement trace. These discovered patterns, represented in the form of logical rules, describe the correlation between network objects. Based on these network patterns and pre-specified domain knowledge, forward and backward inference can be triggered to access global views, predict network status, fire control actions, and diagnose reported problems. Figure 3 illustrates the general approach using learning and inference in network management. Unlike an expert system with only pre-specified domain knowledge, the proposed management system has, in addition, learning ability to augment its knowledge regarding the specific managed network.

Figure 4 is an abstract information flow model of our management systems. EDBs (Extensional Databases) are actually the standard object-oriented MIBs. They represent the basic facts about configuration, traffic/performance measurements, and events/alarms of local nodes. Each network node has an associated EDB which is its local view about the network. IDB (Intensional Database), located at a management site, is defined as the deductive closure of EDBs with logical rules. That is, IDB contains virtual objects defined on the physical objects in EDBs. Access to IDB will be transformed into access to EDBs. This is the same concept as in relational databases where views are virtual relations defined on physical relation tables. EDB and IDB are both deductive database terminologies [Ull88]. The difference is that now IDB is defined on *distributed* EDBs. IDB, including overall configuration and inter-object relationships, embodies the global views of the network. Extracted from IDB, HDB (Historical Database) is the temporal historical database which encode time in the network trace. Network patterns are learned from HDB and stored in PKB (Pattern Knowledge Base). DKB (Domain Knowledge Base) is pre-specified problem solving and general relationship knowledge. Note that only EDBs are standardized; all the others are management application dependent.

A logical rule in IDB/PKB/DKB has the generic form: IF X THEN Y, where X is its *body* part and Y is its *head* part. A body or head part has one or more than one *formula* which can represent the status of a network object or an action to update an object's status. A detailed definition of logical rules in IDB/PKB/DKB is given in the next section.

Each network pattern, represented as a logical rule in PKB, describes a correlation between the attributes of network objects. These correlations are extracted from HDB where selected attributes are logged according to the specific management application. Since this extraction is a statistical process, a probability is associated with each logical rule to show how strong this pattern is.

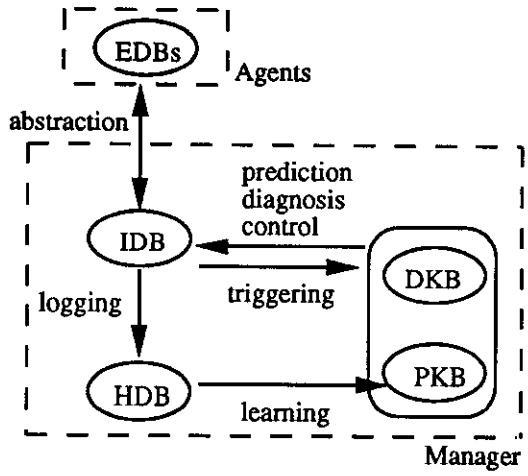


Figure 4: Abstract Model of Information Flow

If the status of network objects satisfies the body part in the rule, the pattern tells us, from the past experience, it is very likely that the status of the network object also satisfies the head part with some probability. This logical rule is thus fired as a forward inference. Forward inference is very suitable for status prediction. If some undesired status of a network object is foreseen to occur, it can further fire some logical rules in DKB and then trigger preventive control actions. On the other hand, if a trouble is reported to the management system (eg. blocking probability of connection 839 is larger than 5%), this object associated with the trouble is matched against the object in the head of rules. If the head is satisfied, the rule is fired as a backward inference and a series of inferences on the formulas in the body can carry on. Finally, the set of residual formulas which can not be further deduced are the possible causes to that trouble. Again, using forward inference on the logical rules in DKB, the remedial control actions can be triggered.

Why Rule-based Systems for Network Management?

After presenting this methodology, let us examine the reasons to adopt rule-based systems for autonomous network management. The following characteristics of network management problems make the rule-based solution desirable:

- Evolving problem-solving knowledge and changing network patterns
- Complex condition matching
- Solution and pattern naturally expressed in IF-THEN rules

New services or solutions are introduced from time to time, which results in updating problem-solving knowledge. Moreover, different network patterns exist in different networks and they may change over time. Since rules are modular pieces of information that are not explicitly directed by

control statements in the program, it is possible to add or remove rules without changing the overall structure of the program or the control flow. In the procedural systems, on the other hand, these changes in problem-solving knowledge may result in modification, recompilation, and reinstallation of the program code. In addition, rule-based systems are powerful in symbolic manipulation by pattern matching between data and rules. A complex situation encoded by many pieces of data can be matched with a set of rules which is then fired to trigger control actions. It is also more common and natural for network domain experts to express their expertise as declarative IF-THEN rules, rather than as procedural algorithms. Besides, the network patterns which represent cause-effect correlations are naturally expressed as logical rules.

4 Induction and Deduction in Network Management

4.1 Terminologies for Logical Rules

As mentioned previously, a rule has the generic form: IF X THEN Y. However, the actual formats and meanings of rules in IDB, PKB, and DKB are different. Here we give the definition of our IDB rule, PKB rule, and DKB rule.

An *IDB rule* is written in the form of Horn clauses, which are statements of the form: "if A_1, A_2, ... , and A_n are true, then B is true." Following the Prolog [SS86] syntax, it is written as

B :- A_1, A_2, ... A_n.

where the formulas B and A_i are *predicates*, e.g. $p(X_1, ..., X_k)$, with a list of arguments. Predicates produce true or false as a result; i.e., they are Boolean-valued functions. A predicate can represent a physical object class stored in EDBs, which is called *EDB predicate*, or a virtual object class defined by IDB rules, which is called *IDB predicate*. IDB rules are used in a backward-chaining fashion to support view abstraction. That is, a query expressed as the predicate B will be transformed into a set of queries/predicates $\{A_i\}$ according to the above IDB rule.

The format of a *PKB rule* is in Horn clauses, with certainty factors, like

Confidence Factor = P% $B \leftarrow A_1, A_2, ... A_n$;

which reads "if A_1, A_2, ..., and A_n are true, B is concluded to be true with probability P." A formula A_i or B is a *condition* that represents the status of a network object (eg., connection status = "closing", 40% $\leq$ link utilization $\leq$ 60%). PKB rules are triggered in a forward-chaining fashion for status prediction. A transformation from PKB rules to DKB rules is required when PKB rules are to be included in the production system for inference.

A *DKB rule* is actually a production rule. It is written in the OPS5 (Official Production

System, version 5) [CW88] syntax as

$(A_1\ A_2\ ...\ A_n \rightarrow B_1\ B_2\ ...\ B_m)$

which reads "if A_1, A_2, ..., and A_n are true, B_1, B_2, ..., and B_m will be executed." Here A_i is a *condition* and B_j is an *action*. DKB rules can be used to invoke control actions by forward inference. They can also emulate backward inference to diagnose problems [BFKM85].

4.2 Induction for Pattern Discovery

Learning is a process of knowledge acquisition. Knowledge can be acquired through taking advice, (i.e., inputing new knowledge directly), problem-solving experience (i.e., remembering the structure of the problem and the methods used to solve it), learning from examples (constructing concept definition from examples), etc [RK91]. Network measurements are themselves examples containing many implicit, network-dependent patterns to be discovered. The inductive learning constructs decision trees from a large number of examples. Each decision tree represents a concept with the following definition [Qui86].

Definition: A concept i includes the function f_i to be approximated, the set of approximators A_i, the domain D_i ($D_i \subseteq$ HDB), on which f_i and the members of A_{ij} are defined, and the confidence factors, CF_i, which is the percentage of examples in D_i that satisfy the following rule:

$f_i(D_i) \in [l'_{ij}, u'_{ij}] \longleftarrow A_{ij}(D_i) \in [l_{ij}, u_{ij}]$ for all attribute j,

where i is the concept index and j is the attribute index. □

As there may be many sets of l', u', l, and u (lower and upper bounds) for a particular set of examples, a set of such rules can be generated from a concept (decision tree). Computational complexities for these learning algorithms are usually exponential in the number of attributes. However, there are steps to reduce complexity by using domain knowledge to restrict the set of attributes and relational structures considered. As learning algorithms are not the main theme of this work, readers are referred to the literature [RK91,Qui86,Win75,Qui87,Pag90,SE92].

In our approach, induction is performed on the management application dependent HDB to generate PKB. The logical rules in PKB model and represent the correlations between attributes in HDB. [GL91B] reports an experiment on interconnected LANs where traffic patterns are learned by a machine learning tool from traffic measurements stored in a HDB implemented as a relational database. The discovered rules can describe traffic patterns in terms of locality, long-term burstiness, correlation, cyclic repetition, and predictability. These patterns can be used for medium-term and long-term performance management.

In addition to traffic patterns, there are many other interesting network patterns. In general, patterns describe inter-object and intra-object relationships. Here an object instance is an example.

We classify the network patterns into the following categories with examples:

- Temporal and geographical traffic distribution

  Confidence Factor = 85%
  20M $\leq$ Traffic $\leq$ 30M
  $\leftarrow$
  11:30AM $\leq$ Time $\leq$ 12:30PM,
  Source = "oahu",
  Destination = "maui";

- Traffic/performance relationships

  Confidence Factor = 90%
  Delay Violation $\geq$ 5%
  $\leftarrow$
  CPU Utilization $\geq$ 60%,
  Network Application Weight $\geq$ 40%;

- Performance correlation between network entities

  Confidence Factor = 80%
  40% $\leq$ Node B Utilization $\leq$ 50%
  $\leftarrow$
  50% $\leq$ Link 1 Utilization $\leq$ 60%,
  35% $\leq$ Link 2 Utilization $\leq$ 50%;

- Hidden causal relationships

  Confidence Factor = 84%
  Node 3 Fails
  $\leftarrow$
  Number of Performance Alarms from Link 1 $\geq$ 10,
  Link 1 Utilization $\leq$ 20%;

4.3 Deduction for Abstraction, Prediction, Control, and Diagnosis

Both preventive and remedial control actions can be taken by network management applications. Preventive control is triggered by problem forecasting based on previous patterns, while remedial control is triggered by network events (performance alarms and device failures). As the manager receives results of the queries to IDB, it passes the configuration status variables to configuration submanager, performance status variables to performance submanager, and event variables to fault

submanager. If any match between the variable values and the body of a rule occurs, the rule is fired and the head part executed. A rule in IDB/DKB/PKB can be fired for four possible purposes:

- Prediction: The forward inference on a PKB rule, given that the rule body is true, forecasts that the rule head will be true.
- Control: The forward inference on a DKB rule triggers the control actions to take when some network phenomena are detected.
- Diagnosis: The backward inference on a DKB or PKB rule can discover the root causes of network events, even when these events are not yet detected.
- Abstraction: The backward inference on an IDB rule transforms an IDB query to EDB query/queries and hence provides view abstraction.

Here are two example inference processes: (i) a process that predicts traffic demands between node X and Y, forecasts performance alarms for link L, and takes actions to reroute some traffic from link L, (ii) a process that diagnoses the received performance alarms, concludes that node Z is malfunctioning, reroutes traffic that passes node Z, and disables node Z.

Backward inference is triggered by events (i.e. only when there are network problems: performance alarms and device failures) and queries (from manager to IDB). However, forward inference is triggered by a set of state variables. The workload on forward inference process can be very high since each state variable will match against each formula in the rule bodies to see if some rules can be fired. Thus, keeping the number of state variables for triggering forward inference small is critical in designing management applications.

5 Distributed Management Architecture

This section and the following two sections describe how the proposed learning and inference schemes work on the standard OSI management platform, the organization of object classes in EDB and rule classes in IDB/PKB/DKB, and the rule-based learning expert system.

Figure 5 shows a management system with a manager and several remote agents. An agent resides on each OSI node and manages its MIB (EDB in our terminology). The manager has submanagers (configuration, performance, and fault inference modules in this case) for specific management functions. Periodically, the manager issues query to IDB, in turn forwarded and translated to EDBs, to get management information and stores it into HDB. The results of query are also passed to submanagers to trigger the inference process on PKB and DKB, if any match

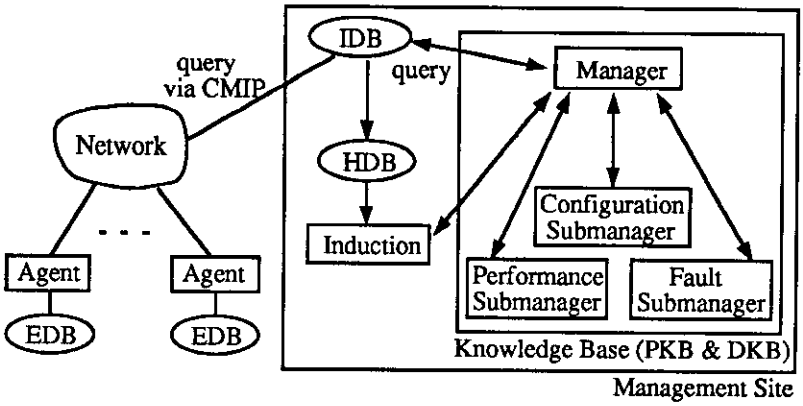


Figure 5: Manager, Submanagers, and Agents

occurs. If control actions are to be fired as a result of the inference, the manager updates the corresponding views in IDB, in turn objects in EDBs, through the sets of methods associated with the objects. These updates on EDBs then propagate to the network entities as control actions. Note that a query via CMIP can be a read as collecting management information or a write (create, delete, modify) as taking control actions.

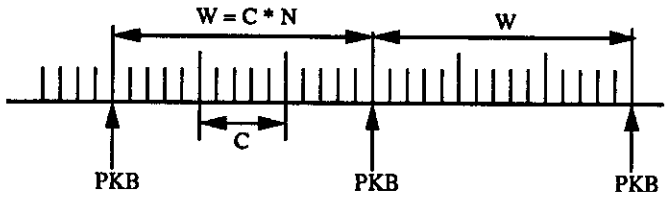


Figure 6: Induction and Query Periods

Induction on HDB is carried out also periodically, but much less often, to renew PKB. A sliding window mechanism is used to maintain the consistency between HDB and PKB. That is, HDB only contains records in the most recent W (window size) query periods. An induction is triggered if PKB was generated W query periods ago. Figure 6 illustrates the relationship between induction period and query period. If C is the number of query periods in a cycle and N is the number of cycles in the window of HDB, $W = C * N$. If there is a temporal repetition in network behavior, cycles exist as network patterns.

Conceptually, management information and knowledge are spread in the layer structure and contained in various databases and knowledge bases:

Layer	Contained in
Control Strategy	DKB
Management Knowledge	DKB and PKB
Object and View Manipulation Rules	IDB
Network Objects	EDB

This is similar to the hierarchical blackboard architecture used in signal-processing expert systems [Hay85]. The control strategy, which is implemented as the manager, decides when to execute the rule sets, which are implemented as the submanagers, in configuration, performance, and fault domains. Usually, this is triggered by either status variables or queries. An inference process on DKB and PKB then accesses the objects of a view in IDB, which in turn accesses the remote objects in EDBs over the network. The hierarchy is organized as Figure 7. The following two subsections describe how to build the views, namely the construction of IDB from the underlying EDBs, and the rule-based management applications based on this infrastructure.

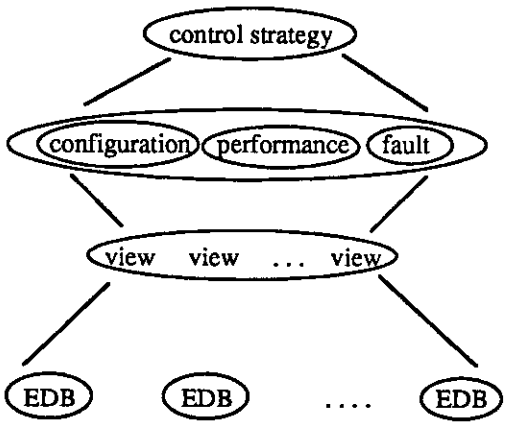


Figure 7: The Blackboard Architecture

6 Management Information Infrastructure

6.1 Object Hierarchy in MIB

Modeling network management information is to map network configuration, performance, and events to objects in EDBs. The *inheritance hierarchy* in Figure 8 represents a simple classification of network object classes where *elements* class has three subclasses: *configurations*, *performances*, and *events*. *Physical entities* class has two subclasses: *nodes* and *links*, etc.

A node's EDB contains only its *local* management information. Figure 9 shows an EDB organized in a *containment hierarchy* and its type declaration. An EDB is an object instance of *nodes*. In addition to its own variable attributes, this *nodes* instance contains a set of *links* instances (for

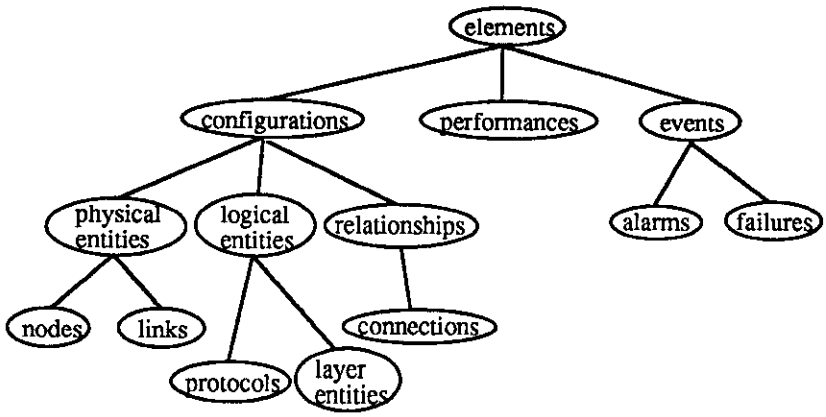


Figure 8: Inheritance Hierarchy

links that are connected to this node), a set of *connections* instances (for connections that pass this node), and a set of *events* instances (for events in which this node is involved). Again, a *links* instance also contains a set of *connections* instances (for connections that pass this link).

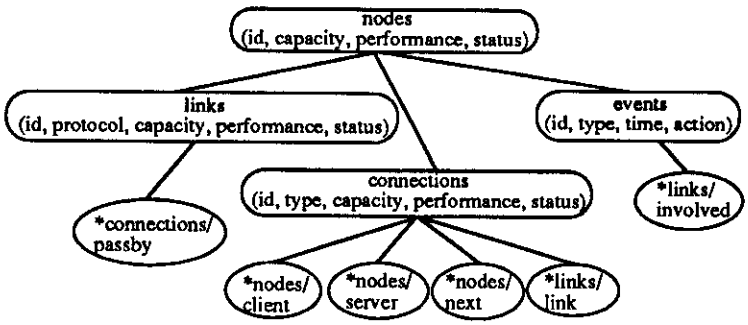
6.2 View Abstraction via Logic Programming

At the management site, what the management applications see is a set of *views*, i.e., a set of IDB predicates. Different sets of views can be defined for different management applications. Each IDB predicate is defined on EDB predicates. The schema at the management site for IDB/EDB predicates and the Prolog implementation to define these IDB predicates are given in Figure 10. Prolog Logic Programming techniques used here can be found in [SS86]. Prolog's recursive programming style, which is not supported in relational query languages, provides us a powerful query interface to unify distributed network management information. Prolog emerges as an attractive query language for relational databases [Ull88]. However, with simple extensions, Prolog can also interface with object-oriented databases [Zan86].

To see how an IDB predicate can be constructed by combining EDB predicates, let us take predicate *connections* as an example:

```
connections(Connid, Type, Capacity, Perfid, Status, Clientid, Serverid,
Nodes, Links)
:- l_connections(Clientid, Connid, Type, Capacity, Perfid, Status,
Clientid, Serverid, _, _),
path(Connid, Clientid, Serverid, Nodes, Links).


path(Connid, End, End, [End], []) :- !.
```



NodeType = RECORDOF(id: int, capacity:int, performance: PerfType, status: int, links: SETOF(LinkType), connections:SETOF(ConnectionType), events: SETOF(EventType));

LinkType = RECORDOF(id: int, protocol:ProtocolType, capacity: int, performance: PerfType, status: int, passby: SETOF(Connection Type));

ConnectionType = RECORDOF(id: int, type: int, capacity: int, performance: PerfType, status: int, client: NodeType, server: NodeType, next: NodeType, link: LinkType);

EventType = RECORDOF(id: int, type: int, time: int, action: ActType, involved: SETOF(LinkType));

PerfType = RECORDOF(id: int, traffic: int, delay: int, loss: int, interval:int);

Figure 9: EDB: A Local MIB

path(Connid, Start, End, [Start | Noderest], [Linkid | Linkrest])
:– l_connections(Start, Connid, _, _, _, _, _, _, Nextid, Linkid),
path(Connid, Nextid, End, Noderest, Linkrest).

For every "Nodeid", predicate *l_connections*(Nodeid, Connid, Type, Capacity, Perfid, Status, Clientid, Serverid, Nextid, Linkid) contains all connections that pass node "Nodeid". (*l* stands for local.) For every such connection, *l_connections* contains "Nextid" and "Linkid" for its next hop (node and link), but doesn't know the whole path. *connections*, constructed from *l_connections*, contains the link lists "Nodes" (all nodes on this connection) and "Links" (all links on this connection). "Nodes" and "Links" are constructed by predicate *path* which takes "Nextid" and "Linkid", starting from the node "Clientid", and inserts them into the link lists "Nodes" and "Links". Note that, in the rule for *connections*, "Nodeid" in *l_connections* is an existential quantifier, which means all nodes can be queried to match with the attributes of *connections*. Similar view construction techniques are used in predicates *links* and *events*. Instead of using recursive predicate *path*, "set_of" constructs are used to construct the link list whose elements satisfy the specified condition. *links* contains "Nodes" (for all nodes connected to this link) and "Events" (for all events involving this link), while *event* contains "Nodes" (for all nodes involved in this event) and "Links" (for all links involved in this event).

Manager's Schema for EDBs:

```
l_nodes(Nodeid, Capacity, Perfid, Status, Links, Conns, Events)
l_links(Nodeid, Linkid, Protocol, Capacity, Perfid, Status,Conns)
l_connections(Nodeid, Connid, Type, Capacity, Perfid, Status, Clientid,
  Serverid, Nextid, Linkid)
l_events(Nodeid, Eventid, Type, Time, Action, Links)
l_performance(Nodeid, Perfid, Traffic, Delay, Loss, Interval)
```

Views in IDB:

```
nodes(Nodeid, Capacity, Perfid, Status, Links, Conns, Events)
links(Linkid, Protocol, Capacity, Perfid, Status, Nodes, Conns, Events)
connections(Connid, Type, Capacity, Perfid, Status, Clientid, Serverid,
  Nodes, Links)
events(Eventid, Type, Time, Action, Nodes, Links)
performances(Perfid, Traffic, Delay, Loss, Interval)
```

View Definitions:

```
nodes(Nodeid, Capacity, Perfid, Status, Links, Conns, Events) :-
l_nodes(Nodeid, Capacity, Perfid, Status, Links, Conns, Events).

links(Linkid, Protocol, Capacity, Perfid, Status, Nodes, Conns, Events) :-
l_links(Nodeid, Linkid, Protocol, Capacity, Perfid, Status, Conns),
set_of(N, (member(Linkid, N_links), l_nodes(N, _, _, _, N_links, _, _)), Nodes),
set_of(E, (member(Linkid, E_links), l_events(_, E, _, _, _, E_links)), Events).

connections(Connid, Type, Capacity, Perfid, Status, Clientid, Serverid, Nodes, Links) :-
l_connections(Clientid, Connid, Type, Capacity, Perfid, Status, Clientid, Serverid, _, _),
path(Connid, Clientid, Serverid, Nodes, Links).

path(Connid, End, End, [End], []) :- !.
path(Connid, Start, End, [Start|Noderest], [Linkid|Linkrest]) :-
l_connections(Start, Connid, _, _, _, _, _, _, Nextid, Linkid),
path(Connid, Nextid, End, Noderest, Linkrest).

events(Eventid, Type, Time, Action, Nodes, Links) :-
set_of(Nodeid, l_event(Nodeid, Eventid, Type, Time, Action, _), Nodes),
set_of(Linkid, (member(Linkid, E_links), l_events(Nodeid, Eventid, Type, Time, Action,
  E_links)), Links).

performances(Perfid, Traffic, Delay, Loss, Interval) :-
l_performances(Nodeid, Perfid, Traffic, Delay, Loss, Interval).
```

Figure 10: Views in IDB

All the predicates mentioned here are the schema definitions at the management site. An access to a predicate of IDB will be converted, by *backward chaining*, to access to predicate(s) of the manager's EDBs, and then transferred, by CMIP queries, to the physical EDBs on network nodes. Thus, a mapping between access to predicates of the manager's EDBs and CMIP queries to physical EDBs must be done at the management site. The attribute "Nodeid" in each EDB predicate is used to identify the network node that contains the object instances.

7 Building Management Applications

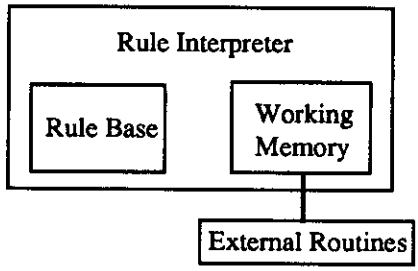


Figure 11: The Rule-based Production System

Based on the constructed IDB, management applications can perform inference on IDB using their knowledge base in DKB and PKB. Data in IDB is matched with the rules in DKB and PKB. Rules can be applied in either direction: forward and backward. The direction corresponds to the type of reasoning and problem-solving strategy. Forward inference is data-driven and bottom-up processing, while backward inference is goal-driven and top-down processing. Prediction and control operations are data-driven, hence forward reasoning. Diagnosis problems are goal-driven, hence backward reasoning.

As shown in Figure 11, there are four components in the rule-based system: working memory, rule base, rule interpreter, and external routines. The rule-based programming language OPS5 is used to describe how to build the management applications. Although OPS5's inference engine is inherently forward, backward inference can be emulated by treating goals as data and using three sets of rules: a set to split goals into subgoals, a set to recognize and solve achievable subgoals, and a set to fuse the results of subgoals. All these rule-based programming techniques can be found in [CW88,BFKM85]. The following paragraphs describe the functionalities and operations of these components in our framework:

A. Working Memory

Periodically, a set of rules is triggered to retrieve data from IDB into working memory. A

working memory element (WME) is a view in IDB. Event WMEs are treated as goals to trigger diagnosis process, while non-event WMEs are data to trigger prediction and control process.

B. Rule Base

In Figure 7, we have three management applications: configuration, performance, and fault. Each management application is associated with a rule cluster. Rules in a cluster is in either DKB or PKB. A rule cluster is conceptually equivalent to a procedure. These rule clusters are scheduled by the control rules, which are at the top level of blackboard architecture.

Rule clusters and control rules together perform the periodical management tasks. A period, a management cycle, starts with retrieving data from IDB into working memory and ends when no more data can trigger the rules. In addition to this synchronous management cycle, there are a set of demon rules to perform asynchronous management. Demon rules are not scheduled by the control rules. They can fire any time when an event WME is detected in working memory. An urgent event like device failure or performance alarm can be handled immediately by demon rules.

C. Rule Interpreter

Basically, the rule interpreter performs a match-select-act cycle to process WMEs. It first matches all WMEs with condition elements in all rules, selects one rule with matching WMEs, and performs actions on the right-hand-side of the chosen rule, and then repeats the cycle.

An event WME is taken as a goal to trigger backward rules to diagnose the root causes of this event. An event WME can be further split into several event WMEs until those event WMEs are root causes. All other WMEs may trigger forward reasoning if a set of WMEs matches the body of a rule.

D. External Routines

The rule-based management applications need to communicate with other management subsystems to access management information and invoke algorithmic routines. To retrieve management information or issue control actions on network entities, queries will be issued to IDB. This is done via external calls to a Prolog program that supports the virtual IDB. Many of the numerical algorithms like bandwidth allocation and path routing are not suitable to be implemented in the rule-based language. They are also implemented as external routines.

8 An Experiment on LAN Environment

The goals of the experiment are to understand the traffic distribution in the environment of interconnected LANs, to test the ability of a learning tool in discovering usual and unexpected traffic

patterns, and to observe the stability of traffic patterns and explore its applicability in performance management.

The stored database will be examined by a machine learning tool called IXL (Induction on eXtremely Large database) [IW88]. IXL is a software tool developed by IntelligenceWare Inc. and made available to us under a joint research project. It combines machine learning and statistics to distill knowledge from large databases. Basically, it constructs topological neighborhoods for database records and then performs generalizations on these neighborhoods to discover rules which show the correlations between attributes in a relation/view [Par89].

Discovered rules which represent traffic patterns, network malfunction, system status will be stored in a knowledge base. A traffic controller The basic approach in this experiment is to monitor the system at the host and network levels. For each fixed period, we summarize the statistics and insert them into a database. After the whole experiment is completed, we apply IXL to the database to generate a set of rules. These rules will reflect the traffic patterns, and more specifically will give us a cause/effect knowledge about such patterns.

The database discovery technique can be applied to a variety of different traffic and network environments. The most obvious situation is that of a real network on which real traffic measurements are collected. In some cases, however, it may be of interest to inject artificial traffic in the network, to simulate one or more applications and to evaluate the traffic patterns resulting by the interaction of such applications. In other cases, the experiment may even be carried out on a computer simulated network environment, with the purpose of studying the effect of events which are difficult to control in a real network environment (e.g. link/node failures, packet loss, overloads, dynamic network reconfiguration, etc).

We describe an experiment on a real, interconnected LAN environment with real traffic. The schema of a set of relations was defined to organize and store the management information. A program was written to process collected measurements and perform data analysis.

8.1 Environment

The experiment is based on the interconnected LAN environment at UCLA Computer Science Department. There are eight Ethernets and one Appletalk interconnected by routers and more than 300 hosts (including mini computers, multi-user or single-user workstations, etc.), many terminal servers, file servers, news servers, and printers (see Figure 12). Most hosts run under UNIX. The transport layer protocol is TCP suite. Different networks are interconnected via IP routers. Because of the structure of the TCP/IP address [Pos81], we can tell which LAN a station belongs to by examining its address. This will help in analyzing the traffic flows between LANs. We monitored the traffic on the backbone Ethernet (i.e. 131.179.128 on Figure 12) which is connected

to the off-department network and to Los Nettos. The monitoring program runs on a SUN-4/280 minicomputer which is attached to the backbone Ethernet.

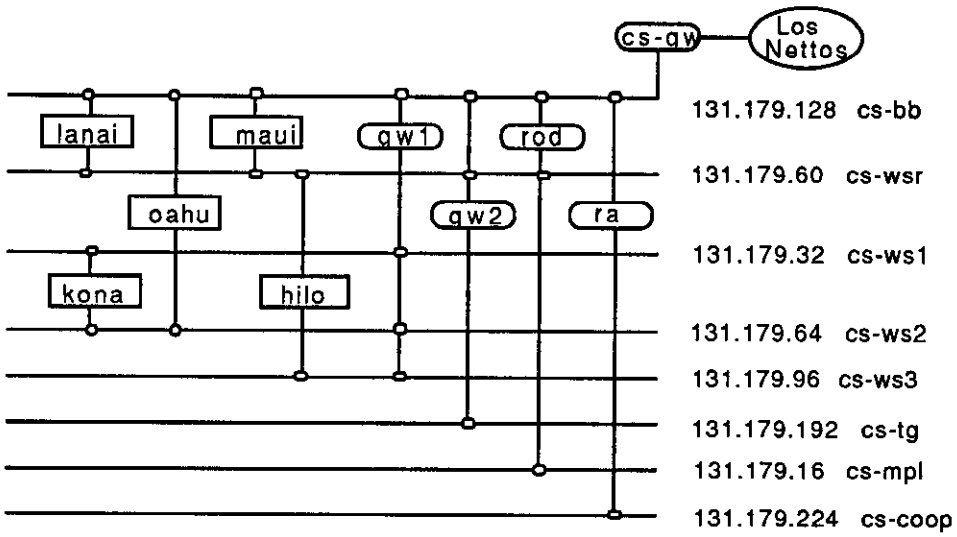


Figure 12: LANs at Computer Science Dept.

The transparent NFS is supported in a way that users can access their own file systems on any host without specifying where they are. This feature accounts for a significant portion of the traffic because of the large amount of file transfers between users' original hosts (or file servers) and current sites. E-mail delivery, remote procedure calls, news reading, file printing, tape backup, human-initiated terminal emulation sessions, and human-initiated file transfers also account for the accumulated traffic amount, in addition to the machine-initiated file transfers mentioned above. It is observed that NFS and window protocol (e.g. X window, SunView) traffic dominates traffic generated by the other protocols like "rlogin", "telnet", and "ftp". With the increasing number of diskless workstations and window users, the profile will become clearer.

8.2 The Traffic Pattern Observer

Figure 13 is the overview of the Traffic Pattern Observer which is composed of several tools integrated by user interface. The major components of the system are Monitors, IXL, and a set of utilities. Monitors will activate a set of tools to monitor traffic and, at the same time, a set of handlers to handle the traffic data generated by those tools. The utilities will provide an interface to the database query language and also contain tools for defining and setting up database. The Database Interface is the one providing transparency of the DBMS (Data Base Management System) used so that we can switch to another DBMS without affecting other components of the system.

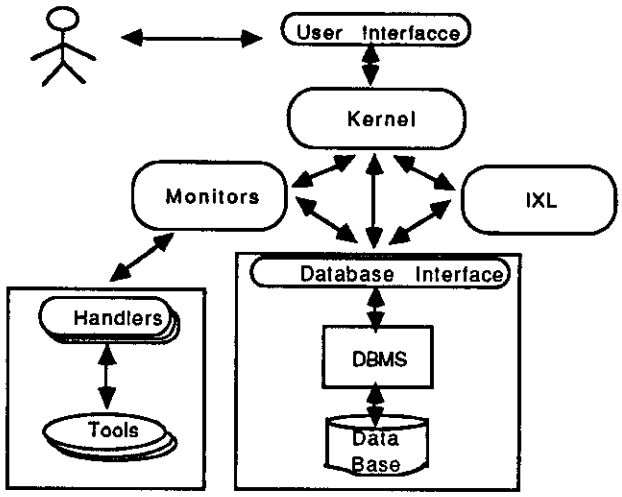


Figure 13: The Traffic Pattern Observer

Before the experiment can be conducted, the database schema must be defined in the DBMS. For each tool, there should be a base table in the database and a handler associated with it so that all of the tool handlers can work concurrently. Also, the structures of the tool handlers depend on the schema specified in the DBMS.

At the conceptual level of the database, the base table schema is fixed; however, the user may have the freedom to supply the view definition which depends on the expected knowledge to be discovered. If the user supplies his/her own view definition, there may be some meaningless results generated by IXL if the view definition has some defects like join of two base tables with no common column. To guarantee a reasonable result, the system will provide a set of view definition for users to choose from.

During the monitoring process, there are a set of processes working concurrently. Some are listening to the Ethernet and pumping the information they have captured, some are processing the pumped data and maintaining the data structures to keep track of the summarized statistics, some are busy with the database interface to insert records into the base tables. The data structures maintained in the handlers are inserted into base tables and purged every period of time. Although there are a lot of process working on this job, which can be considered as a considerable overhead to the system, only processes which fetch the host information by some remote execution mechanism will transmit packets on the Ethernets. The influence on the results concerning network traffic can be small. However, the performance of the local host may be affected. If the experiment program is run on a dedicated workstation, there will be no influence to other hosts. Furthermore, the communication overhead can be minimized if we run IXL at the same site where the database is located.

It is expected that we may have new tools for monitoring some other activities. If a new tool needs to be included, the system maintainer needs to do the following:

1. Supply a handler associated with the new tool and insert a new entry in the tool table of Monitors subsystem which may invoke the new tool during monitoring process.
2. Insert new base table definitions into the original schema and those new base tables will contain the information available via the tool.
3. Create new view definition associated with the new base tables and those new view definitions will be new alternatives for users to choose from before IXL is invoked.

To monitor an Ethernet, we use a UNIX network maintenance tool, etherfind. Etherfind detects all the packets transmitted on the Ethernet. It dumps the IP headers and puts a timestamp on them. In the IP header, the following fields are particularly interesting to this experiment: source address, destination address, number of bytes, protocol type, and fragmentation flag. We wrote a program to handle the headers dumped by etherfind and couple them together by a pipeline.

Several buffer arrays are used to monitor the current active communication entities. When the etherfind handler receives a packet header, it checks the buffer arrays to see if the entity exists. If a match is found, the corresponding entry is updated. Otherwise, a new entry is created. This buffer is swept periodically, for each time slot T, and each entry is either promoted to file entry or flushed. In order to reduce the storage requirements while capturing the most significant traffic components, we promote only those entries which percentagewise contribute most to the traffic in that time slot. The entry with largest contribution will be promoted first and then the second one, etc. When the promoted entries capture P% of total traffic, the promotion process stops and the buffers are flushed. A new time slot then begins. This promotion process for data reduction is shown in Figure 14. A typical experiment lasts one to several days. In our experiment, the capture ratio was set to $P = 80$.

The structure of the buffer space is identical to the schema for storing promoted entries in a relational database. Four tables are defined for this experiment. "Summary" just summarizes the total traffic and connections. Traffic is also classified into local (within the local Ethernet), incoming (coming from remote LANs), outgoing (going to remote LANs), and transit (both source and destination are not on this LAN). (see Figure 15) "Connections" keeps track of the current active communicating node pairs. The traffic amount and type for each pair are recorded. "BLANs" is similar to "Connections" except it is between source LAN and destination LAN, instead of between nodes. "Sources" traces the source nodes which contribute to the traffic on the monitored Ethernet. Here are the definitions of these tables: (note: the fields with underline are keys.)

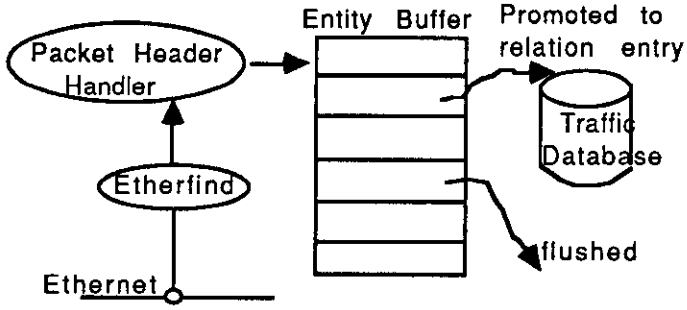


Figure 14: Promotion Process for Data Reduction

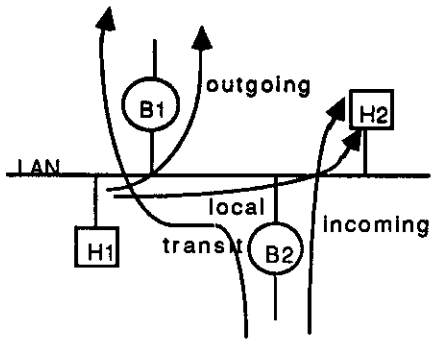


Figure 15: LAN's Internal and External Traffic

SUMMARY :

Slot : start time of this time slot
Bytes : # of Kbytes successfully transmitted
TCP : % of tcp traffic transmitted
UDP : % of udp traffic transmitted
Connections : number of node connections
Promoted : %connections being promoted
Captured : %traffic contributed by promoted connections
LocalTraffic : %traffic with source and dest on this LAN
IncomingTraffic : %traffic with only dest on this LAN
OutgoingTraffic : %traffic with only source on this LAN
TransitTraffic : %traffic with source and dest not on this LAN

SOURCES :

Slot : start time of this time slot
Source : source station address
Bytes : #Kbytes transmitted from this station
Percentage : %traffic from this station
Nodetype : local or remote node

CONNECTIONS :

Slot : start time of this time slot
Source : source station address
Dest : destination station address
Bytes : #Kbytes transmitted between this pair
Percentage : %traffic between this pair
Type : (local, incoming, outgoing, transit)

BLANS :

Slot : start time of this time slot
SourceLAN : source LAN address
DestLAN : destination LAN address
Bytes : #Kbytes transmitted between this LAN pair
Percentage : %traffic between this LAN pair
Type : (local, incoming, outgoing, transit)

The traffic measurements are transferred from SUN-4/280 to PC DOS disks via IBM RT after the monitoring process is completed. IXL then runs on those relational tables in an IBM PC/AT. Each IXL run takes from several minutes to several hours, depending on the size of relational tables and various discovery parameters set in IXL. Also, the number of generated rules depends heavily on the settings of discovery parameters. By properly setting these parameters, we can direct IXL to find the traffic distribution and patterns we need. In this experiment, we monitored for 5 days. The sizes of generated tables are from 300 to 5000 tuples. The running times of IXL on these tables are between 10 minutes to 5 hours. The numbers of discovered rules are between 10 to 100. Typically, several rounds of experiments are required in order to adjust table size, IXL running time, and focus of discovery.

IXL also supports the definition of "concepts", which are virtual fields derived from other existing fields. These "concepts" can reduce the running time of IXL and help focusing the discovery process. In our experiment, we define the concept "Traffic" to classify the levels of traffic volume. For example:

Traffic = very high if Bytes $\geq$ 10 Mbytes;
Traffic = high if 10 Mbytes > Bytes $\geq$ 5 Mbytes;
Traffic = medium if 5 Mbytes > Bytes $\geq$ 1 Mbytes;
Traffic = low if 1 Mbytes > Bytes;

IXL has a set of parameters which tailor its performance to the user's need [IW88]. Major discovery parameters include the following: maximum number of clauses in rules (an upper limit for the length of a rule), minimum number of records (a lower limit for the number of records involved in forming a rule), minimum confidence in rules (a lower limit for the confidence in a rule), maximum margin of error (an upper limit for the error involved in estimating the confidence in a rule), minimum percentage of database (a lower limit for the fraction of the database involved in forming a rule), minimum significance (a measurement of the quality of a range in terms of how the distribution of values in that range varies from the rest of the database where 0 means that almost all ranges are considered and 100 means that only highly significant ranges are considered), minimum generality (a upper boundary for the range sizes determined by IXL), maximum generality (a lower boundary for the range sizes), generality increments (an indicator of the number of ranges between the maximum and minimum generality parameters where 0 means only two ranges, maximum and minimum generality, are considered and 100 means up to 20 ranges are considered), and interest level (user's interest in the effect that a field has on the goal).

8.3 Experimental Results

The experiment includes two sample runs on the tables "summary" (288 tuples) and "BLANs" (2151 tuples) where numbers of rules found are 43 and 53, respectively. The IXL running time is 13 minutes for "summary" and 1 hour 50 minutes for "BLANs". In these sample runs, we focus on the discovery of relationship between traffic volume and other fields. Thus, we make the defined concept "Traffic" as our goal attribute in the rules to be discovered. Of those discovered rules, some are particularly interesting to us:

CF=85
"traffic" = "very high"
IF
"0:55" $\leq$ "timeslot" $\leq$ "1:35"
AND
"91%" $\leq$ "outgoing" $\leq$ "94%" ;

CF=95
"traffic" = "high"
IF
"12:28" $\leq$ "timeslot" $\leq$ "13:53"
AND
"sourceLAN" = "131.179.64"
AND
"destLAN" = "131.179.192" ;

CF (confidence ratio) in the rule means the percentage of records satisfying the goal among the records satisfying the conditions of the rule. The first rule is discovered for "summary" where "very high" means volume is larger than 10 Mbytes in a 5-minute slot. This rule indicates that from 0:55AM to 1:40AM, outgoing traffic accounts for around 90larger than 10 Mbytes/slot. Actually, this happens when the system is backing up its file system to tapes every morning around 1:00AM to 2:00AM. That most traffic is outgoing implies that the backup tape is not on the backbone Ethernet. Indeed, the backup machine is "131.179.32.11", a SUN-4/280 on another LAN. We believe a peer rule will be discovered if we run the same experiment also on the LAN where the backup machine resides, except that "outgoing" becomes "incoming". Since the traffic volume caused by tape backup varies each day, there is a high degree of fluctuation in the periods of tape backup as shown in Figure 16. However, we can still find the correlation and the cycle.

The second rule above is discovered for "BLANs" where "high" means volume is larger than 500 Kbytes/slot but smaller than 1 Mbytes/slot between source and destination LANs. This rule

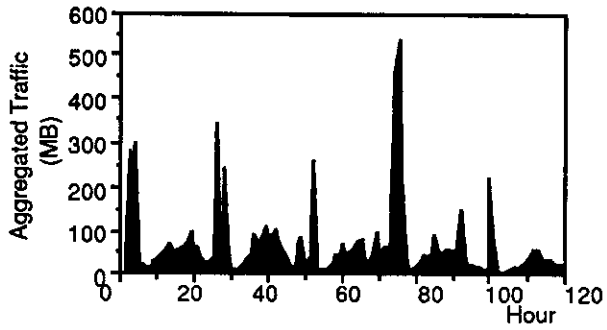


Figure 16: Traffic Cycles

means that the traffic volume from LAN "131.179.64" to LAN "131.179.192" between 12:28PM and 13:58PM is between 500 Kbytes/slot and 1 Mbytes/slot. This type of rule can be very useful in understanding the traffic distribution with respect to topology and time. It captures the traffic distribution in a three-dimensional traffic matrix.

More than 80% of the traffic is contributed by less than 10% of communicating pairs, ie. traffic is not uniformly distributed. It is essential to capture this distribution in order to optimize the network configuration. A temporal cycle exists in the traffic distribution. Being able to keep track of the distribution cycle will enable the dynamic configuration management which tunes the network dynamically. If we consider the burstiness in terms of different time scale, the inter-slot burstiness (long-term) is reflected by the cycle, while the intra-slot burstiness (short-term) can be approximated by a Batch Poisson or Markov-Modulated Poisson process. This is due to the fact that we summarize the measurements for each slot, thus some details within the slot are lost and can only be approximated by a stochastic process.

For the discovery process, tuning the learning parameters to fit the need of the application is not a trivial task. In order to have a reasonable set of discovered rules, IXL parameters must be carefully set. For example, too few rules will be generated if the minimum confidence is too high. The sizes of the ranges for "timeslot" in the rules will be too small if the generality increments are set to zero (default). One of the limitation of IXL is that it is not suitable to learn the correlation within the numerical values. It must rely on the proper classification of the numerical domain to reduce the number of unique values to be handled.

9 GlobeView/LEN: a Management Framework and a Case Study

In this section, we present the implementation of GlobeView which realizes the proposed global view concept. A case study on applying a learning expert, LEN (Learning Expert for Networks), to ATM network topology tuning is also reported.

GlobeView is implemented in Prolog. It is aimed to serve as an interface through which queries can be issued to the distributed network entities from the network management applications. It abstracts the distributed management information and provides an integrated view for the management applications. Basically, there are four types of queries: create, delete, retrieve, and update. In the mean time, a query can be a local query which only accesses an individual node or a global query which accesses a set of correlated nodes. A design of management information base, which is EDB using our terminologies, is described. This EDB includes predicates for nodes, links, events, and VPs (Virtual Paths) in ATM networks. Based on this EDB, a set of global views in IDB are implemented as virtual predicates by backward chaining rules. We also implement global queries to create/delete physical EDB entities and retrieve/update the attributes of IDB predicates. Many sophisticated management operations can be implemented as global queries. Examples on sizing the capacities of VPs are given.

As an automatic and adaptive management system, LEN issues queries as management operations to GlobeView to create or delete network entities, retrieve management information, and update attributes of network entities. In our case study, we focus on evaluating the performance gain by learning traffic locality patterns to tune the network topology dynamically. Traffic locality data is generated from a simulator and learned by a machine learning tool to produce the PKB that contains logical rules of traffic locality. These rules drive the tuning process on a timeslot-by-timeslot basis. In each timeslot, the topology is optimized to reduce call blocking probability according to the learned locality patterns. The simulated traffic matrix is then applied to the tuned topology to evaluate call blocking probability. The result is compared with the non-tuned one.

9.1 GlobeView

We first describe our EDB design and then the construction of IDB predicates. The technique to build global queries as management operations is illustrated by several examples. The detailed implementation is listed in Appendix.

9.1.1 EDB and IDB

Our schema of EDB predicates basically follow the ones described in section 6. We rename the predicate *l_connection* to *l_VP* to represent a VP in ATM networks. We add an attribute, *Residual*, in predicates *l_nodes* and *l_links* to represent how much capacity is left for further allocation. Thus, we have the following five EDB predicates:

l_nodes(nodeid, capacity, perfid, status, links, vps, events, residual).
l_links(nodeid, linkid, protocol, capacity, perfid, status, vps, residual).

l_vps(nodeid, vpid, type, capacity, perfid, status, clientid, serverid, nextid,
linkid).
l_events(nodeid, eventid, type, time, action, links).
l_performances(nodeid, perfid, traffic, delay, loss, interval).

Corresponding to the EDB predicates, we have five basic IDB predicates. These IDB predicates are the basic global views. All other global information can be derived from them. They also facilitate the construction of other sophisticated global queries. Defining predicates *nodes* and *performance* is straightforward as:

```
nodes(Nodeid, Capacity, Perfid, Status, Links, VPs, Events, Residual)
   :- l_nodes(Nodeid, Capacity, Perfid, Status, Links, VPs, Events, Residual).

performance(Perfid, Traffic, Delay, Loss, Interval)
   :- l_performances(_,Perfid, Traffic, Delay, Loss, Interval).
```

Predicate *links* requires the built-in predicate *set_of* to construct a set of elements that satisfy another user-specified predicate. The list elements in attributes *Nodes*, *VPs*, and *Events* are constructed in this way. A complete implementation is as follows:

```
links(Linkid, Protocol, Capacity, Perfid, Status, Nodes, VPs, Events,
Residual)
   :- l_links(_, Linkid, Protocol, Capacity, Perfid, Status, _, Residual),
   nodes_on_link(Linkid,Nodes),
   vps_on_link(Linkid,VPs),
   events_on_link(Linkid,Events).

nodes_on_link(Linkid,Nodes)
   :- setof(Nodeid, find_nodes(Nodeid,Linkid), Nodes).

find_nodes(Nodeid,Linkid)
   :- l_nodes(Nodeid,_,_,_, N_links,_,_,_),
   member(Linkid,N_links).

vps_on_link(Linkid,VPs)
   :- setof(VPid, find_vps(VPid,Linkid), VPs).

find_vps(VPid,Linkid)
```

```
    :- l_vps(_, VPid,_,_,_,_,_,_,_,Linkid).

events_on_link(Linkid,Events)
    :- setof(E, find_events( E , Linkid), Events).

find_events( E, Linkid )
    :- l_events(_,E,_,_,_, E_links),
    member(Linkid, E_links).

find_events( [],_).
```

Using the same approach, we can build predicate *events* as follows:

```
events(Eventid, Type, Time, Action, Nodes, Links)
    :- l_events(_,Eventid, Type, Time, Action,_),
    setof(Nodeid, detect_nodes(Nodeid,Eventid), Nodes),
    setof(Linkid, detect_links(Linkid,Eventid), Links).

detect_nodes(Nodeid,Eventid)
    :- l_events(Nodeid, Eventid,_,_,_,_).

detect_links(Linkid,Eventid)
    :- l_events(_, Eventid,_,_,_, E_links),
    member(Linkid, E_links).
```

Predicate VPs, on the other hand, requires a recursive predicate, which is implemented as *path*, to construct the path of this VP. The path starts from the client node, through intermediate links and nodes, to the server node. A complete implementation is as follows:

```
vps(VPid, Type, Capacity, Perfid, Status, Clientid, Serverid, Nodes, Links)
    :- l_vps(Clientid, VPid, Type, Capacity, Perfid, Status, Clientid, Serverid,
    _,_),
    path(VPid, Clientid, Serverid, Nodes, Links).

path(VPid, Start, End, [Start | Noderest], [Linkid | Linkrest])
    :- l_vps(Start, VPid,_,_,_,_,_,_,Nextid, Linkid),
    path(VPid, Nextid, End, Noderest, Linkrest).
```

```
path(_, X,X , [X], []) :- !.
```

Next, we show how to write management operations as global queries which are built from the basic IDB predicates presented here.

9.1.2 Global Queries as Management Operations

We now give examples on the global queries for retrieval and update. For management information retrieval, it is sufficient to access the basic five IDB predicates and five EDB predicates. However, writing a long predicate to access a piece of information is inefficient. Besides, we may want to make only a subset of information accessible to a specific manager or management application. Thus, we can write a set of predicates for frequently accessed information. For example, we may want to know the set of nodes that are attached to a specified link, which is a point-to-point or broadcast media, the set of nodes on the path of a specified VP, the set of links on the path of a specified VP, or the capacity of a specified VP. These can be easily provided by writing the following predicates:

```
nodes_of_link(Linkid,Nodes)
   :- links(Linkid,_,_,_,_,Nodes,_,_,_).

nodes_on_vp(VPid,Nodes)
   :- vps(VPid,_,_,_,_,_,_,Nodes,_).

links_on_vp(VPid,Links)
   :- vps(VPid,_,_,_,_,_,_,_,Links).

capacity_of_vp(VPid,Capacity)
   :- vps(VPid,_,Capacity,_,_,_,_,_,_).
```

Updating management information through global queries is not as simple as global information retrieval. It is different from updating the attributes of an EDB predicate in a single network node, which can be done by accessing that EDB predicate alone. We need to issue a *single* global update query which changes the attributes of *many* correlated EDBs. For example, we may want to disable, by a single query, all nodes and links that are currently involved in a serious congestion event. We may need to allocate more bandwidth to a VP which is suffering higher delay or loss rate. We now take the second example to illustrate how to implement a global update query for this.

Suppose that the query $update\_capacity\_VP(VPid, Capacity)$ is used to update the capacity of a VP to *Capacity*. If the new capacity is smaller than the current capacity, we can go ahead to

retrieve the nodes and links on this VP, add the released capacity to each of these nodes and links, and update the capacity of this VP. If the new capacity is larger than the current one, we need to first retrieve the nodes and links on this VP and check if each of these nodes and links has enough residual capacity to allocate more capacity to this VP. If the answer is yes for all nodes and links, their capacities are subtract by the amount of the extra allocation and the capacity of this VP is updated. If not, the query does nothing and returns a message. The first case, shrinking a VP, is handled by the rule:

```
update_capacity_vp(VP,Capacity)
  :- capacity_of_vp(VP,Capacity1),
  Capacity1 > Capacity,
  Residual is Capacity - Capacity1,
  nodes_on_vp(VP,Nodes),
  links_on_vp(VP,Links),
  change_nodes(Nodes,Residual),
  change_links(Links,Residual),
  change_vp(VP,Capacity).
```

The second case, augmenting a VP, is handled by the rule:

```
update_capacity_vp(VP,Capacity)
  :- capacity_of_vp(VP,Capacity1),
  Residual is Capacity - Capacity1,
  nodes_on_vp(VP,Nodes),
  links_on_vp(VP,Links),
  !,
  check_nodes(Nodes,Residual),
  !,
  check_links(Links,Residual),
  change_nodes(Nodes,Residual),
  change_links(Links,Residual),
  change_vp(VP,Capacity).
```

The referred predicates in the above two rules are further implemented as follows. For checking if there is enough residual capacity, we have

```
check_nodes([Node | Left],Residual)
```

```
    :- !,
    check_node(Node,Residual),
    check_nodes(Left,Residual).

check_nodes([],_).

check_node(Node,Residual)
    :- residual_of_node(Node,Available),
    !,
    Available > Residual.

check_links([Link | Left],Residual)
    :- !,
    check_link(Link,Residual),
    check_links(Left,Residual).

check_links([],_).

check_link(Link,Residual)
    :- residual_of_link(Link,Available),
    :- !,
    Available > Residual.
```

For changing capacities of nodes and links, we have

```
change_nodes([Node | Left],Residual)
    :- change_node(Node,Residual),
    change_nodes(Left,Residual).

change_nodes([],_).

change_node(Node,Residual)
    :- l_nodes(Node,A1,A2,A3,A4,A5,A6,Old),
    retract(l_nodes(Node,_,_,_,_,_,_,_)),
    New is Old - Residual ,
    assertz(l_nodes(Node,A1,A2,A3,A4,A5,A6,New)).

change_links([Link | Left],Residual)
```

```
    :- change_link(Link,Residual),
    change_links(Left,Residual).

change_links([],_).

change_link(Link,Residual)
    :- l_links(A1,Link,A2,A3,A4,A5,A6,Old),
    retract(l_links(A1,Link,A2,A3,A4,A5,A6,Old)),
    New is Old - Residual ,
    assertz(l_links(A1,Link,A2,A3,A4,A5,A6,New)),
    fail.

change_link(_,_).
```

To change capacity of the VP, we simply use *retract* and *assertz* as follows.

```
change_vp(VP,New)
    :- l_vps(Nodeid,VP,Type,Capacity,Perfid,Status,
    Client,Server,Nextid,Linkid),
    retract(l_connections(Nodeid,VP,Type,Capacity,Perfid,
    Status,Client,Server,Nextid,Linkid)),
    assertz(l_vps(Nodeid,VP,Type,New,Perfid,
    Status,Client,Server,Nextid,Linkid)),
    fail.

change_vp(_,_).
```

With this powerful predicate *update_capacity_VP*, operations for bandwidth allocation can be carried out by issuing a single query. This query is then automatically transformed into many local queries to retrieve and update many pieces of information. And all these are done transparently to the management application which is in charge of the bandwidth allocation task. The implementation described in this section has been tested and fully working. Further extensions to other management tasks can be done in a similar way.

9.2 LEN: Learning Experts for Networks

Most of the congestion and flow control procedures for conventional networks can not be applied to ATM networks where nodes tend to become the bottlenecks and propagation delay dominates

other delays. Congestion (eg. call blocking, cell loss) is inevitable if there is a mismatch between offered traffic pattern and network topology. This problem can be alleviated by dynamically tuning the topology to traffic pattern. With the technology of digital cross-connect systems (DCS), a broadband packet-switched ATM network can be dynamically reconfigured. The embedded logical topology can be derived from the original physical topology by establishing express pipes between distant nodes. Express pipes, circuit-switched pipes for packet-switched traffic, reduce store and forward delay and nodal processing overhead, which in turn reduces blocking and loss probabilities. Given the traffic demand matrix, the routing of express pipes and the allocation of bandwidth to such pipes, ie. embedded topology, can be determined to optimize the GOS (Grade of Service) [GMP89].

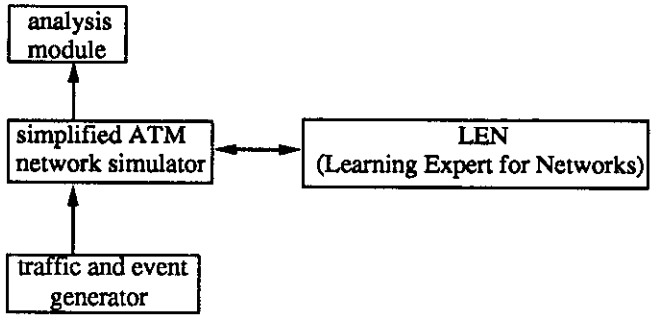


Figure 17: Case Study on ATM Networks

In this case study, we will demonstrate how to learn traffic patterns and tune the topology to the discovered patterns, and the performance improvement with this scheme. Figure 17 illustrates the case study. Traffic is generated according to our model which incorporates the parameters for adjusting locality, burstiness, correlation, cyclic repetition, and predictability. The simulated traffic is fed into a simplified ATM network simulator and the performance results are evaluated by an analysis module. The management system LEN (Learning Expert for Networks) performs the management tasks by monitoring the network simulator, learning traffic patterns, and trigger actions to tune the topology. A comparison is made between the performance results for systems with and without LEN. As LEN is still under development, the inference process on the rule base is now done manually.

9.2.1 ATM Network Model: Configuration, Traffic, and Operation

A. Configuration

Each node v_i has a switching capacity K_i bps. Each link l_{ij} between two nodes v_i and v_j has capacity C_{ij} bps. Similarly, each express pipe p_{ij} between nodes v_i and v_j has capacity B_{ij} bps.

Nodes, links, and express pipes have other attributes. Their schemata are the same as those in Figure 10. Express pipes are represented as the view *vps* in that schema. In LEN, three types of configuration WMEs are created: node, link, and pipe. Their attributes are again the same as the ones in Figure 10.

B. Traffic

To generate a relational traffic table HDB, a base matrix describing bandwidth requirement is generated first, and then N cycles of traffic matrix are generated. A filtering process is imposed on the traffic table to capture only significant traffic components and reduce the stored information volume. (i) and (ii) are the specifications for traffic generation and filtering:

(i) Base matrix generation:
Input: Percentages of heavy, medium, light traffic pairs H%, M%, L%
Output: Mean level matrix of bandwidth requirement $MEAN(i,j)$,
variance level matrix of bandwidth requirement $VAR(i,j)$

(ii) Traffic database generation:
Input: $MEAN(i,j)$, $VAR(i,j)$, discrete usage habit curve $U(t)$,
maximum promotion ratio p%, maximum capture ratio c%,
number of cycles generated N
Output: Relational Traffic Table HDB(hour, source, dest, bandwidth)

HDB will be the input to our machine learning tool IXL (Induction on eXtremely Large databases) [IW88]. The learning result is a set of PKB rules.

The defined traffic model reflects the following characteristics: locality, correlation, burstiness, predictability. Conceptually, $MEAN(i,j)$, $VAR(i,j)$, and $U(t)$ are used to randomly generate a 3-D bandwidth requirement matrix $Fij(t)$, where i is source, j is destination, and t is time slot. In the mean time, p% and c% (p% of communicating pairs contributing c% of total traffic) are used as criteria for the promotion process to capture significant part of collected traffic measurements [GL91B]. N cycles of traffic measurements, HDB, will serve as a base for predicting the traffic distribution of next cycle.

(iii) Induction for Traffic Patterns:
The inducted PKB rule

Confidence Factor = P%
LF $\leq$ bandwidth $\leq$ UF
$\leftarrow$

START $\leq$ slot $\leq$ END,
sourcenode = SRC,
destnode = DST;

means the bandwidth requirement of a particular node pair during several continuous slots is between two values with probability P%. The function of IXL is to find out when and how much traffic is flowing from Src to Dest, both in terms of ranges. The establishment, at *Start*, and release, at *End*, of pipes are discrete events.

PKB is an abstract of HDB. It represents the patterns in the past N cycles. According to these inducted patterns, the topology of ATM network will be tuned with some express pipes established. Each such inducted rule will be automatically transformed into the following two rules and then included into OPS5 rule base:

```
(p performance!predict-and-create-traffic-WME
        (subtask ^name performance)
        (time ^slot START)
        (node ^name SRC)
        (node ^name DST)
      ⟶
        (make traffic ^predicted-rate (compute-rate P LF UF)
                      ^from START ^until END
                      ^source SRC ^dest DST))

(p performance!delete-traffic-WME
        (subtask ^name performance)
        (time ^slot END)
        { <demand>
        (traffic ^until END)}
      ⟶
        (remove <demand>))
```

C. Operation

LEN is responsible for tuning the topology according to PKB. A management cycle in LEN includes the following steps:

- Retrieve data into WM: call external Prolog program to issue queries to IDB
- Predict traffic demand: create traffic WMEs

- Handle traffic WMEs: create pipe WMEs
- Reconfigure topology: call external Prolog program to write pipe views to IDB

At the beginning of a time slot, LEN checks if the current slot matches any "START" or "END" entry in PKB. If any match occurs, four possible actions can be taken: establish new pipes, augment existing pipes, shrink existing pipes, and release existing pipes. In the above four cases, LEN can create new traffic WMEs, modify or delete existing traffic WMEs. These traffic WMEs will match with a set of rules to create, modify, or delete pipe WMEs. Another set of rules then match these pipe WMEs to call external routines to physically access the express pipes. The algorithm for pipe bandwidth allocation and pipe routing is described in [GMP89]. The following is an example rule to create a pipe WME from a new traffic WME:

```
(p performance!create-pipe-WME-from-traffic-WME
        (subtask ^name performance)
        { <demand>
        (traffic ^processed? nil
                ^predicted-rate <flow> ^source <src>
                ^dest <dst> ^delay-requirement <delay>)}
        (status ^congestion-level <system-load>)
    ⟶
        (modify <demand> ^processed? YES)
        (make pipe ^bandwidth
                (compute-bandwidth <flow> <delay> <system-load>)
                ^source <src> ^dest <dst>))
```

9.2.2 Performance Gain by Learning Traffic Patterns

While tuning the ATM network in each time slot according to PKB of the previous N cycles, the traffic for the next cycle is generated and applied to the tuned network to compute the connection blocking probability. In the mean time, this probability for the non-tuned network is also computed for comparison.

In our simulation run, the ATM network contains 8 nodes and 15 links. The simulated traffic has a base traffic matrix $MEAN(i,j)$ of Table 1.(a). The traffic locality is about 9/80 (9% communicating pairs contributing 80% of total traffic) as shown in Table 1.(b). One day is a cycle which is divided into 48 time slots. During the period of five cycles, a total of 675 traffic pairs are promoted (with capture ratio set to 80%) and logged into HDB. The induction process takes 2 hours and 49

minutes and generates a PKB containing 74 rules. In the ATM network simulation, we manually change the topology for each time slot according to PKB and then apply another cycle of traffic. As we complete LEN implementation, this will be done automatically.

	A	B	C	D	E	F	G	H
A	0	4	1	2	1	4	0	2
B	4	0	48	36	18	0	0	24
C	1	48	0	1	1	373	353	0
D	2	36	1	0	4	0	2	398
E	1	18	1	4	0	3	225	4
F	4	0	373	0	3	0	23	2
G	0	0	353	2	225	23	0	2
H	2	24	0	398	4	2	2	0

(a) Traffic Matrix

Capture(%)	Promotion(%)
26	3.5
50	7
73	11
88	14
91	18
93	22
95	25

(b) Traffic Locality

Table 1. Traffic for an 8-Node Network

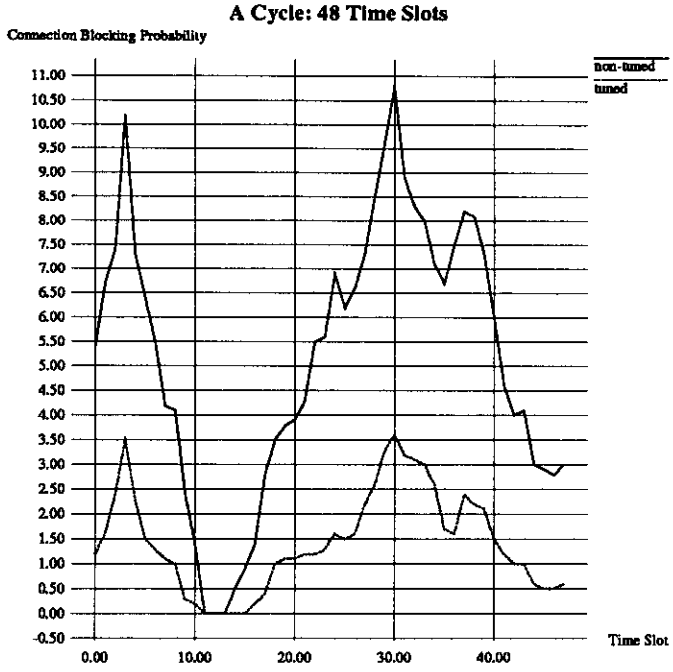


Figure 18: Connection Blocking Probability: Non-Tuned and Tuned

The resulting connection blocking probabilities for non-tuned and tuned networks are compared in Figure 18. The averaged connection blocking probabilities (weighted by traffic volume) under the given load are 6.54% and 1.71% for the ATM network without and with tuning, respectively. The improvement is significant, especially when the network is heavily loaded.

9.3 Discussions

Inference, as a thinking process on given facts by logical rules, is to find the facts that are not explicitly stated in the knowledge base. That is, the deductive closure, K^+, can be derived from the fixed knowledge base, K, but no more than that. Learning, on the other hand, can expand K^+ by adding facts or rules to K. In our proposed framework, knowledge related to the underlying network is learned to capture network patterns and refine the pre-specified domain knowledge. The learning systems have more advanced abilities than non-learning systems in performance and fault management which require understanding of traffic patterns and knowledge of causality.

The proposed scheme is aimed to operate on the standard platform of MIBs and CMIP. Two main contributions are the global view abstraction and the integration of learning and inference for autonomous management applications. The case study on ATM logical topology tuning shows significant improvement when dynamic traffic patterns are captured to drive the tuning process. The implementation of the LEN (Learning Expert for Networks) is now in progress. Other performance and fault management applications will be built in LEN.

10 Conclusions and Future Work

Network management controls resource allocation and maintains quality services which require sophisticated systems in the increasingly complicate networks. Our contributions are reviewed in the following sections.

10.1 View Abstraction via Logic Programming

The *information distribution strategy* remains an unsolved problem after the open-networking community settled on the standard MIB and CMIP. What management applications need is a set of global information to trigger the overall network resource allocation. We proposed to use the backward chaining *logic programming* practice to construct *virtual global* information from *physical local* information scattered around the *distributed* network. Indeed, there are many ways to construct global information. Logic programming approach turns out to be a systematic and elegant solution.

A global view is defined as a piece of information that is not available on any individual MIB but can be constructed from a set of MIBs. A global view can be, for example, the set of nodes that are connected to a broadcast link, the set of nodes and links that are on the path of a particular VP in ATM networks, the set of nodes and links that are involved in a congestion event, the set of VPs that have performance alarms, etc. All these views can be abstracted from MIBs by backward

chaining rules. These views serve as the *windows* through which the management information infrastructure can be accessed. Different management applications access different views for their management tasks.

10.2 Learning Experts for Autonomous Networks

Since we do not have models for all activities in the network, we proposed to use heuristic methods to discover patterns related to the managed network. Management applications take these discovered patterns to adjust their solutions for resource allocation and problem diagnosis.

Many expert systems have been built to diagnose network faults. Domain knowledge regarding how to solve network problems is encoded into these systems. In addition to the domain knowledge, our approach reasons on pattern knowledge to solve network prediction, control, planning, and diagnosis problems. Our experiment on learning traffic patterns shows that this approach is feasible. As machine learning becomes more powerful, we can foresee that the learning experts will be more appealing in autonomous network management.

10.3 GlobeView/LEN Implementation

Finally, we report the implementation of GlobeView/LEN. GlobeView is implemented in Prolog. An EDB schema and an IDB schema are defined. View abstraction for IDB is done by the recursive and set_of constructs in Prolog. GlobeView is implemented as a database query system which supports queries to create, delete, retrieve, and update our IDB. We demonstrated the results of these queries on a manually constructed network configuration database.

As we do not have a network where a MIB is supported by each node, GlobeView is implemented solely as a query system working on a single database instead of a set of distributed MIBs. However, GlobeView can be modified to invoke the CMIP protocol to access remote MIBs.

In LEN implementation, we applied the learning expert approach to ATM network logical topology tuning. Dynamic traffic locality is learned into logical rules which drive the tuning algorithm. When a new locality pattern rule is fired, LEN tries to tune the topology to fit this new request, by establishing a new pipe and adjusting the capacities of the old pipes, without changing the routing the old pipes. If such an attempt fails due to the significant change in pipe capacities, a reconfiguration is enforced. The inference part of LEN, in OPS5, is not finished. Thus, the inference is now done manually in our case study. In this study, we observe significant improvement in call blocking probability, especially when the blocking is serious in the non-tuned network.

10.4 Future Work

As MIB and CMIP are soon to be available, we would expand our GlobeView to operate on the real, distributed environment. GlobeView can be used as a stand-alone query system for network managers and also as an interface to the underlying information infrastructure for the management applications.

Issues need to be addressed include how to transport the EDB predicates to the remote MIBs by the CMIP protocol, how to extend the IDB program to operate on the object-oriented EDBs in an efficient way, the delay constraints in constructing the global views for various management tasks, the architecture to allow managers or management systems on multiple nodes, etc. Indeed, building a workable GlobeView on the real environment is not a trivial task.

LEN implementation shall proceed. Once it is done, LEN should interact with GlobeView as a management application with a speciality in topology tuning and reconfiguration, if GlobeView operates on an ATM network. In the mean time, other management applications can be developed for configuration management, performance management, fault management, etc. However, all these require a significant amount of implementation efforts.

Several issues need to be answered before developing a new management application as a learning expert. What kinds of problem does this management application solve? What kinds of patterns need to be learned? Is the current machine learning systems capable of capturing these patterns? How well will the network perform with this learning expert compared to the one without this mechanism?

Appendix: GlobeView Implementation

% Manager's Schema for EDBs:

```
l_nodes(nodeid, capacity, perfid, status, links, conns, events,
risidual_capacity).
l_links(nodeid, linkid, protocol, capacity, perfid, status, conns,
risidual_capacity).
l_connections(nodeid, connid, type, capacity, perfid, status, clientid,
serverid, nextid, linkid).
l_events(nodeid, eventid, type, time, action, links).
l_performances(nodeid, perfid, traffic, delay, loss, interval).
```

```
% Definition of Views in IDB:

% Views for nodes:

  nodes(Nodeid, Capacity, Perfid, Status, Links, Conns, Events, Residual) :-
    l_nodes(Nodeid, Capacity, Perfid, Status, Links, Conns, Events, Residual).

  active_nodes(Nodeid, Capacity, Perfid, Status, Links, Conns,
  Events, Residual) :-
    nodes(Nodeid, Capacity, Perfid, Status, Links, Conns, Events,
    Residual),
    Status is on.

% Views for links:

  links(Linkid, Protocol, Capacity, Perfid, Status, Nodes, Conns,
  Events, Residual) :-
    l_links(_, Linkid, Protocol, Capacity, Perfid, Status, _, Residual),
    nodes_on_link(Linkid,Nodes),
    conns_on_link(Linkid,Conns),
    events_on_link(Linkid,Events).

  nodes_on_link(Linkid,Nodes) :-
    setof(Nodeid, find_nodes(Nodeid,Linkid), Nodes).

  find_nodes(Nodeid,Linkid) :-
    l_nodes(Nodeid,_,_,_, N_links,_,_,_),
    member(Linkid,N_links).

  conns_on_link(Linkid,Conns) :-
    setof(Connid, find_connections(Connid,Linkid), Conns).

  find_connections(Connid,Linkid) :-
    l_connections(_,Connid,_,_,_,_,_,_,_,Linkid).

  events_on_link(Linkid,Events) :-
    setof(E, find_events( E , Linkid), Events).
```

```
find_events( E, Linkid ) :-
    l_events(_,E,_,_,_, E_links),
    member(Linkid, E_links).

find_events( [],_).

active_links(Linkid, Protocol, Capacity, Perfid, Status, Nodes, Conns,
Events, Residual) :-
    links(Linkid, Protocol, Capacity, Perfid, Status, Nodes, Conns,
    Events, Residual).

% Views for connections:

connections(Connid, Type, Capacity, Perfid, Status, Clientid, Serverid,
Nodes, Links) :-
    l_connections(Clientid, Connid, Type, Capacity, Perfid, Status, Clientid,
    Serverid,_,_),
    path(Connid, Clientid, Serverid, Nodes, Links).

% path predicate constructs the whole path from Client to Server:

path(Connid, Start, End, [Start|Noderest], [Linkid|Linkrest]) :-
    l_connections(Start, Connid,_,_,_,_,_,_, Nextid, Linkid),
    path(Connid, Nextid, End, Noderest, Linkrest).

path(_, X,X , [X], []) :- !.

active_connections(Connid, Type, Capacity, Perfid, Status, Clientid, Serverid,
Nodes, Links) :-
    connections(Connid, Type, Capacity, Perfid, Status, Clientid, Serverid,
    Nodes, Links),
    Status is on.

% Views for events:

events(Eventid, Type, Time, Action, Nodes, Links) :-
    l_events(_,Eventid, Type, Time, Action, _),
    setof(Nodeid, detect_nodes(Nodeid,Eventid), Nodes),
```

```
    setof(Linkid, detect_links(Linkid,Eventid), Links).

  detect_nodes(Nodeid,Eventid) :-
    l_events(Nodeid, Eventid,_,_,_,_).

  detect_links(Linkid,Eventid) :-
    l_events(_, Eventid,_,_,_, E_links),
    member(Linkid, E_links).

% Views for performance:

  performance(Perfid, Traffic, Delay, Loss, Interval) :-
    l_performances(_,Perfid, Traffic, Delay, Loss, Interval).

% Basic predicates used in these Views:

  member(X , [X|_]):- !.

  member(X , [_|Rest]) :-
    member(X , Rest).

  length([],0).

  length([_|Xs],s(N)) :-
    length(Xs,N).

% This query program has two major parts: retrieve and update.
% (1) retrieve commands
% To construct and retrieve local or global informatiion of the network.
% (2) update commands
% To update local or global information due to some network situation.
% (1) The retrieve commands:

% configuration:

% query on nodes:

  residual_of_node(Nodeid,Residual) :-
```

```
    l_nodes(Nodeid,_,_,_,_,_,_,_,Residual).

  capacity_of_node(Nodeid,Capacity) :-
    l_nodes(Nodeid,Capacity,_,_,_,_,_,_).

  status_of_node(Nodeid,Status) :-
    l_nodes(Nodeid,_,_,Status,_,_,_,_).

  links_to_node(Nodeid,Links) :-
    l_nodes(Nodeid,_,_,_,Links,_,_,_).

  num_links_to_node(Nodeid,Number) :-
    links_to_node(Nodeid,Links),
    length(Links,Number).

  connection_on_node(Nodeid,Conns) :-
  l_nodes(Nodeid,_,_,_,_,Conns,_,_).

  num_connection_on_node(Nodeid,Number) :-
    connection_on_node(Nodeid,Conns),
    length(Conns,Number).

  event_on_node(Nodeid,Events) :-
    l_nodes(Nodeid,_,_,_,_,_,Events,_).

  performance_of_node(Nodeid,Traffic,Delay,Loss,Interval) :-
    l_nodes(Nodeid,_,Perfid,_,_,_,_,_),
    performance(Perfid,Traffic,Delay,Loss,Interval).

% query on links:

  residual_of_link(Linkid,Residual) :-
    links(Linkid,_,_,_,_,_,_,_,Residual).

  protocol_of_link(Linkid,Protocol) :-
    links(Linkid,Protocol,_,_,_,_,_,_,_).

  capacity_of_link(Linkid,Capacity) :-
```

```
    links(Linkid,_,Capacity,_,_,_,_,_,_).

status_of_link(Linkid,Status) :-
    links(Linkid,_,_,_,Status,_,_,_,_).

performance_of_link(Linkid,Traffic,Delay,Loss,Interval) :-
    l_links(_,Linkid,_,_,Perfid,_,_,_),
    performance(Perfid,Traffic,Delay,Loss,Interval).

nodes_of_link(Linkid,Nodes) :-
    links(Linkid,_,_,_,_,Nodes,_,_,_).

num_nodes_of_link(Linkid,Number) :-
    nodes_of_link(Linkid,Nodes),
    length(Nodes,Number).

connection_of_link(Linkid,Conns) :-
    links(Linkid,_,_,_,_,_,Conns,_,_).

num_connection_of_link(Linkid,Number) :-
    connection_of_link(Linkid,Conns),
    length(Conns,Number).

event_of_link(Linkid,Events) :-
    links(Linkid,_,_,_,_,_,_,Events,_).

% query on connections:

type_of_connection(Connid,Type) :-
    connections(Connid,Type,_,_,_,_,_,_,_).

capacity_of_connection(Connid,Capacity) :-
    connections(Connid,_,Capacity,_,_,_,_,_,_).

performance_of_connection(Connid,Traffic,Delay,Loss,Interval) :-
    connections(Connid,_,_,Perfid,_,_,_,_,_) ,
    performance(Perfid,Traffic,Delay,Loss,Interval).
```

```
client_of_connection(Connid,Client) :-
    connections(Connid,_,_,_,_,Client,_,_,_).

server_of_connection(Connid,Server) :-
    connections(Connid,_,_,_,_,_,Server,_,_).

nodes_on_connection(Connid,Nodes) :-
    connections(Connid,_,_,_,_,_,_,Nodes,_).

num_nodes_on_connection(Connid,Number) :-
    nodes_on_connection(Connid,Nodes),
    length(Nodes,Number).

links_on_connection(Connid,Links) :-
    connections(Connid,_,_,_,_,_,_,_,Links).

num_links_on_connection(Connid,Number) :-
    links_on_connection(Connid,Links),
    length(Links,Number).

% query on events:

type_of_event(Eventid,Type) :-
    events(Eventid,Type,_,_,_,_).

time_of_event(Eventid,Time) :-
    events(Eventid,_,Time,_,_,_).

action_on_event(Eventid,Action) :-
    events(Eventid,_,_,Action,_,_).

nodes_in_event(Eventid,Nodes) :-
events(Eventid,_,_,_,Nodes,_).

links_in_event(Eventid,Links) :-
    events(Eventid,_,_,_,_,Links).

% The update commands:
```

```
% The first part is used for updating the local data.

  u_capacity_of_node(Nodeid,New_capacity) :-
    l_nodes(Nodeid,Old_capacity, A1, A2, A3, A4, A5, A6),
    retract(l_nodes(Nodeid,Old_capacity,_,_,_,_,_,_)),
    assertz(l_nodes(Nodeid,New_capacity, A1, A2, A3, A4, A5, A6)).

  set_on_node(Nodeid) :-
    l_nodes(Nodeid,_,_,on,_,_,_,_),
    write('it is on').

  set_on_node(Nodeid) :-
    l_nodes(Nodeid,A1,A2,off,A3,A4,A5,A6),
    retract(l_nodes(Nodeid,_,_,_,_,_,_,_)),
    assertz(l_nodes(Nodeid,A1,A2,on,A3,A4,A5,A6)).

  set_off_node(Nodeid) :-
    l_nodes(Nodeid,_,_,off,_,_,_,_),
    write('it is off').

  set_off_node(Nodeid) :-
    l_nodes(Nodeid,A1,A2,on,A3,A4,A5,A6),
    retract(l_nodes(Nodeid,_,_,_,_,_,_,_)),
    assertz(l_nodes(Nodeid,A1,A2,off,A3,A4,A5,A6)).

  u_performance_of_node(Nodeid,Traffic,Delay,Loss,Interval) :-
    performance_of_node(Nodeid,Traffic,Delay,Loss,Interval),
    write('the same performacnce').

  u_performance_of_node(Nodeid,Traffic,Delay,Loss,Interval) :-
    l_nodes(Nodeid,_Perfid,_,_,_,_,_),
    retract(performance(Perfid,_,_,_,_)),
    assertz(performance(Perfid,Traffic,Delay,Loss,Interval)).

% The second part is the global update.

% To update connection capacity.
```

```
% If we want to decrease the capacity,
% Retrive the nodes and links on the path, add the released capacity to
%each of these nodes and links, and update capacity of this connection.

  u_capacity_of_connection(Connect,Capacity) :-
    capacity_of_connection(Connect,Capacity1),
    Capacity1 > Capacity,
    Residual is Capacity - Capacity1,
    nodes_on_connection(Connect,Nodes),
    links_on_connection(Connect,Links),
    change_nodes(Nodes,Residual),
    change_links(Links,Residual),
    change_connection(Connect,Capacity).

% If we want to increase the capacity, first retrieve the nodes and links on this
% connection. Check whether available capacity exits on these nodes and links.
% If yes, for all nodes and links, substrct by the amount of extra allocation and
% update the capacity of this connection.

  u_capacity_of_connection(Connect,Capacity) :-
    capacity_of_connection(Connect,Capacity1),
    Residual is Capacity - Capacity1,
    nodes_on_connection(Connect,Nodes),
    links_on_connection(Connect,Links),
    !,
    check_nodes(Nodes,Residual),
    !,
    check_links(Links,Residual),
    change_nodes(Nodes,Residual),
    change_links(Links,Residual),
    change_connection(Connect,Capacity).

% To increase the capacity of a connection to its maximum,
subject to the limit along the path.

  max_of_connection(Connect,Capacity) :-
    capacity_of_connection(Connect,Old),
```

```
   nodes_on_connection(Connect,Nodes),
   links_on_connection(Connect,Links),
   !,
   full_node(Nodes,Residual1),
   full_link(Links,Residual2),
   choose(Residual1,Residual2,Residual),
   Capacity is Residual + Old,
   change_nodes(Nodes,Residual),
   change_links(Links,Residual),
   change_connection(Connect,Capacity).

full_node([Node|Left],Residual) :-
   !,
   residual_of_node(Node,R1),
   full_node(Left,R2),
   choose(R1,R2,Residual).

   full_node([],10000000).

full_link([Link|Left],Residual) :-
   !,
   residual_of_link(Link,R1),
   full_link(Left,R2),
   choose(R1,R2,Residual).

full_link([],1000000).

choose(R1,R2,Residual) :-
   R1 > R2,
   Residual is R2.

choose(R1,_,R1).

check_nodes([Node|Left],Residual) :-
   !,
   check_node(Node,Residual),
   check_nodes(Left,Residual).
```

```
check_nodes([],_).

check_node(Node,Residual) :-
   residual_of_node(Node,Available),
   !,
   Available > Residual.

check_links([Link|Left],Residual) :-
   !,
   check_link(Link,Residual),
   check_links(Left,Residual).

check_links([],_).

check_link(Link,Residual) :-
   residual_of_link(Link,Available),
   !,
   Available > Residual.

% The following codes are used for updating local data.

change_nodes([Node|Left],Residual) :-
   change_node(Node,Residual),
   change_nodes(Left,Residual).

change_nodes([],_).

change_node(Node,Residual) :-
   l_nodes(Node,A1,A2,A3,A4,A5,A6,Old),
   retract(l_nodes(Node,_,_,_,_,_,_,_)),
   New is Old - Residual,
   assertz(l_nodes(Node,A1,A2,A3,A4,A5,A6,New)).

change_links([Link|Left],Residual) :-
   change_link(Link,Residual),
   change_links(Left,Residual).

change_links([],_).
```

```
change_link(Link,Residual) :-
   l_links(A1,Link,A2,A3,A4,A5,A6,Old),
   retract(l_links(A1,Link,A2,A3,A4,A5,A6,Old)),
   New is Old - Residual,
   assertz(l_links(A1,Link,A2,A3,A4,A5,A6,New)),
   fail.

change_link(_,_).

change_connection(Connect,New) :-
   l_connections(Nodeid,Connect,Type,Capacity,Perfid,Status,
   Client,Server,Nextid,Linkid),
   retract(l_connections(Nodeid,Connect,Type,Capacity,Perfid,
   Status,Client,Server,Nextid,Linkid)),
   assertz(l_connections(Nodeid,Connect,Type,New,Perfid,
   Status,Client,Server,Nextid,Linkid)),
   fail.

change_connection(_,_).
```

References

[BFKM85] Brownston, L., R. Farrell, E. Kant, N. Martin, *Programming Expert Systems in OPS5: An Introduction to Rule-based Programming,* Addison-Wesley, 1985.

[BS84] Buchanan, B. G. and E. H. Shortliffe, *Rule-based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project,* Addison-Wesley, Massachusetts, 1984.

[CDFS88] Case, J. D., J. R. Davinm, M. S. Fedor, and M. L. Schoffstall, *A Simple Network Management Protocol,* RFC 1067, SRI Int., August 1988.

[CPW89] Cassel, L. N., C. Partridge, and J. Westcott, *Network Management Architectures and Protocols: Problems and Approaches,* IEEE Journal on Selected Areas in Communications, Vol. 7, No. 7, September 1989.

[CW88] Cooper, T. A. and N. Wogrin, *Rule-based Programming with OPS5,* Morgan Kaufmann, 1988.

[EEM89] Ericson, E. C., L.T. Ericson, and D. Minoli, editors, *Expert Systems Applications in Integrated Network Management,* Artech House, 1989.

[FB85] Feldman, J. A. and D. H. Ballard, *Connectionist Models and Their Properties,* Cognitive Science, 6, 205-254, 1985.

[For81] Forgy, C. L., *OPS5 User's Manual,* Technical Report CMU-CS-81-135, Dept. of Computer Science, Carnegie-Mellon University, 1981.

[GMP89] Gerla, M., J. S. Monteiro, R. Pazos, *Topology Design and Bandwidth Allocation in ATM Nets,* IEEE Journal on Selected Areas in Communications, Vol. 7, No. 8, October 1989.

[GL91A] Gerla, M. and Y. D. Lin, *Network/Intelligence: An Experiment on Interconnected LANs,* AAAI Workshop on Knowledge Discovery in Databases, Anaheim, July 1991.

[GL91B] Gerla, M. and Y. D. Lin, *Network Management Using Database Discovery Tools,* Proceedings of the 16th Conference on Local Computer Networks, Minneapolis, October 1991.

[Goy91] Goyal, S., *Knowledge Technologies for Evolving Networks,* in Proceedings of the IFIP TC6/WG6.6 Second International Symposium on Integrated Network Management, January, 1991; also in Integrated Network Management, II, I. Krishnan, et al., editors, North-Holland, 1991.

[HBRD93] Haritsa, J. R., M. O. Ball, N. Roussopoulous, A. Datta, *Design of the MANDATE MIB,* in Proceedings of International Symposium on Integrated Network Management, San Francisco, April 1993.

[Hay85] Hayes-Roth, B., *A Blackboard Architecture for Control,* Artificial Intelligence, 26:255-321, 1985.

[IE88] IEEE, *Special Issue on AI Applications to Telecommunications,* Journal on Selected Areas in Communications, June, 1988.

[IW88] Intelligence Ware Inc., *IXL: The Machine Learning System, User's Manual,* Intelligence Ware Inc., 1988.

[ISO90A] ISO/IEC DIS 10165-1, *Information Technology - Open Systems Interconnection - Structure of Management Information - Part 1: Management Information Model,* ISO, Geneva, Switzerland, June 1990.

[ISO90B] ISO 9596, *Information Technology - Open Systems Interconnection - Common Management Information Protocol Specification,* ISO, Geneva, Switzerland, May 1990.

[ISO90C] ISO/IEC DIS 10040, *Information Processing Systems - Open Systems Interconnection - Systems Management Overview,* ISO, Geneva, Switzerland, 1990.

[Lie88] Liebowitz, J., Editor. *Expert System Applications to Telecommunications,* John Wiley, New York, 1988.

[Lew93] Lewis, L., *A Case-Based Reasoning Approach to the Resolution of Faults in Communications Networks,* in Proceedings of International Symposium on Integrated Network Management, San Francisco, April 1993.

[Lin93] Lin, Y. D., *Network Traffic Patterns: Models and Heuristics,* Ph.D. Dissertation, University of California, Los Angeles, June 1993.

[LG92] Lin, Y. D. and M. Gerla, *A Framework for Learning and Inference in Network Management,* IEEE GLOBECOM, Orlando, Florida, 1992.

[LG93] Lin, Y. D. and M. Gerla, *Induction and Deduction for Autonomous Networks,* To appear in IEEE Journal on Selected Areas in Communication in late 1993.

[LTHG93] Lin, Y. D. , T. C. Tsai, J. Huang, and M. Gerla, *HAP: A New Model for Packet Arrivals,* To appear in Proceedings of ACM SIGCOMM '93, San Francisco, September 1993; also to appear in Journal of High Speed Networks in late 1993.

[MBL93] Mazumdar, S., S. Brady, D. W. Levine, *Design of Protocol Independent Management Agent to Support SNMP and CMIP Queries,* in Proceedings of International Symposium on Integrated Network Management, San Francisco, April 1993.

[MR88] McCloghrie, K., and M. Rose, *Management Information Base for Network Management of TCP/IP-based internets,* RFC 1066, August 1988.

[MR90] McCloghrie, K. and M. Rose, *Management Information Base for Network Management of TCP/IP-based Internets,* RFC 1156, Internet Standard, May 1990.

[McD82] McDermott, J., *R1: A Rule-based Configurer of Computer Systems,* Artificial Intelligence, 19, 39-88, 1982.

[MCK+89] Minton, S., J. G. Carbonell, C. A. Knoblock, D. R. Kuokka, O. Etzioni, and Y. Gil, *Explanation-based Learning: A Problem Solving Perspective,* Artificial Intelligence, 40, 63-118, 1989.

[Mit78] Mitchell, T. M., *Version Spaces: An Approach to Concept Learning,* PhD thesis, Stanford University, Stanford, 1978.

[Mos83] Mostow, D. J., *Machine Transformation of Advice into a Heuristic Search Procedure,* In Machine Learning, An Artificial Intelligence Approach, ed. R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, Tioga Press, Palo Alto, 1983.

[Pag90] Pagallo, G. M., *Adaptive decision tree algorithms for learning from examples,* Ph.D. thesis, UCSC-CRL-90-27, University of California, Santa Cruz, 1990.

[PAR89] Parsaye, K., et al., *Intelligent Databases: Object-Oriented, Deductive, Hypermedia Technologies,* p.404-415, John Wiley, New York 1989.

[Pea88] Pearl, J., *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference,* Morgan Kaufmann, 1988.

[Qui86] Quinlan, J. R., *Induction of Decision Trees,* Machine Learning, 1, pp.81-106, 1986.

[Qui87] Quinlan, J. R., *Generating Production Rules from Decision Trees,* in Proceedings of the 10th International Joint Conference on Artificial Intelligence, pp. 304-307, Morgan Kaufmann, 1987.

[RK91] Rich, E. and K. Knight, *Artificial Intelligence, second edition,* p.447-484, McGraw-Hill, Inc., 1991.

[Ros90] Rose, M., *Management Information Base for Network Management of TCP/IP-based Internets - MIB II,* RFC 1158, Internet Standard, May 1990.

[SS90] Saavedra-Barrera, R. and A. J. Smith, *Performance Prediction by Benchmark and Machine Analysis* Report No. UCB/CSD 90/607, December 1990.

[Sho76] Shortliffe, E. H., *Computer-based Medical Consultations: MYCIN,* Elsevier, New York, 1976.

[SE92] Sleeman, D. and P. Edwards, editors, *Machine Learning: Proceedings of the Ninth International Workshop,* Morgan Kaufmann, 1992.

[SS86] Sterling, L. and E. Shapiro, *The Art of Prolog: Advanced Programming Techniques,* MIT Press, 1986.

[Ter92] Terplan, K., *Communication Networks Management, second edition,* Prentice-Hall, New Jersey, 1992.

[Ull88] Ullman, J. D., *Principles of Database and Knowledge-base Systems, Volume I,* p. 11-12, p.82-87, p. 100-101, Computer Science Press, 1988.

[WK81] Weiss, S. M. and C. A. Kulikowski, *Expert Consultation Systems: The EXPERT and CASNET Projects,* Machine Intelligence, Report 9, no. 3, 1981.

[Win75] Winston, P. H., *Learning Structural Descriptions from Examples,* In The Psychology of Computer Vision, ed. P. H. Winston, McGraw-Hill, New York, 1975.

[Zan86] Zaniolo, C., *Prolog: A Database Query Language for All Seasons,* Expert Database Systems, Proceedings From the First International Workshop, L. Kerschberg, Editor, Benjamin/Cummings Publishing, 1986.