Management of Distributed Systems

Y. C. Shim and C.V. Ramamoorthy

Computer Science Division
Dept. of Electrical Engineering and Computer Science
University of California
Berkeley, CA 94720

ABSTRACT

Effective utilization of distributed systems requires many management facilities such as resource management, performance analysis, etc. But all these facilities share three common basic services: collecting information, detecting assertions, and taking actions. Instead of duplicating these basic services in all the management facilities, we introduce a logically centralized tool called the Distributed System Manager (DSM) which performs the three basic services. Each of the management facilities becomes a client to the DSM and registers to the DSM a request of the form <assertion, action> if it wants the action to be taken when the assertion is detected. The DSM monitors the distributed system to collect information using sampling probes and tracing probes and stores the collected information in a history database. The DSM also detects the moment when the assertion in a registered request becomes satisfied and then takes the corresponding action which can be answering clients' queries to system status stored in the history database, notifying the detection of assertion, and exercising direct control over the system. In this paper we discuss some of the issues involved in the design of the DSM. The overall structure of the managed system and the DSM is given. Then data modeling of the managed system, the mechanisms to collect information, and the representation scheme of the information in the history database are described. Lastly the specification language for assertions based on the classical temporal logic is presented.

1. INTRODUCTION

It is well understood that distributed systems offer opportunities for attaining high performance, fault-tolerance, information sharing, etc. But all these desirable goals cannot be achieved unless there is a suitable environment for the distributed system. Some of the facilities provided by this environment are:

1. Resource management - keeps track of available resources in the system and allocates resources efficiently to requests from the clients.
2. Performance analysis - observes the performance of the system, detects the performance degradation, and locates performance bottleneck.
3. Behavior analysis - monitors the execution behavior of the programs and locates errors.

4. Fault management - detects, diagnoses, and recovers from the system faults.
5. Security management - finds any suspicious attempts and behavior in the system and takes actions to counteract them.

Although the purposes of these facilities are different from each other, all of them need following three types of basic services:

1. Collecting information - collects information such as status of the system, occurrence of primitive events, etc. and stores some of them in the database.
2. Detecting assertions - detects the moment when a certain assertion (e.g., a certain host gets overloaded) becomes satisfied.
3. Taking actions - takes actions like answering clients' queries to the system status, notifying the detection of assertions, and exercising direct control over the system.

Realizing that duplication of these three basic services in every system facility is a waste of time and space, we introduce a logically centralized monitoring and control tool called the Distributed System Manager (DSM). The DSM monitors the system to collect information, detects assertions, and takes actions. Each of the facilities introduced above becomes a client to the DSM and registers to the DSM a request of the form <assertion, action> if it wants the action to be taken when the assertion is detected.

Some of the functional issues involved in the development of the DSM are as follow.

1. Information collection - We should understand what kinds of entities or objects exist in the managed system, how they are characterized, and how they are related with each other. Then we should identify what information we want to collect and how we can collect it. The types of collected information fall into two categories, an entity's attribute which comprises the system status and a primitive event occurring in an entity. An attribute can have a static value like the machine type of a host or be a time-varying quantity like the load average of a host. A primitive event can be an operation provided by an entity as a service like a process creation, login, etc. or be an occurrence of an error like a login failure. Finally we have to develop a method to represent and store this infor-
2. Request management - A client's request is a pair <assertion, action>. First a specification language for assertions should be developed along with an efficient and intelligent detection algorithm. The specification language should have enough expressive power so that it can represent facts, events, and any combination of them with a temporal qualification. A fact is a property of the system that holds over stretches of time, e.g., that the load average of a certain host is higher than 5. An event is something happening in the system, e.g., that the load average of a host increases from 3 to 5 or that a certain person logs in a certain host. More mathematically rigorous approaches to handling time will be given in a later section. Then we need a specification language for actions and mechanisms to invoke those actions in a distributed environment.

In two areas, the management of computer networks and the monitoring/debugging of distributed programs, some or all of the above issues have been studied at varying levels of detail. A management system sought in a large computer network, which usually is an internet consisting of many heterogeneous subnetworks, assists network operators to monitor, analyze, and control complex communication networks by providing tools to gather real-time data on network performance and traffic characteristics, diagnose communications problems, and reconfigure networks to meet changing needs and environmental conditions[1, 2, 3]. A managed communication network is considered to be a collection of entities like gateways, hosts, etc. Entity is characterized by a set of system states, control actions which can be invoked to change the system states, and event which is a state change in the system and may be reported. Among these system states are collected and stored in the database. To this managed communication network the management system provides three types of services: data manipulation services, event reporting services, and direct control services. In the management of communication networks, the data modeling of the managed system and the system architecture, data structure, and required functions of the manager have been studied. But all the discussions are at the high level and the details like data collection, assertion specification and detection, and action management remain to be pursued further.

The Network Event Manager (NEM) developed by Chen[4] is a tool used to manage computer networks. The managed system is considered to be a collection of entities and relationships exist among entities. Attributes of entities and relationships are collected using sampling probes and stored in a history database. Assertions are defined using the attribute values and checked against the history database periodically. Actions are taken when an assertion is detected. This system has many novel ideas like entity-relationship model of the system, the system status in a history database, the assertion specification and detection, and the action management. But NEM does not have tracing probes which collect the occurrences of primitive events, the assertion specification allows only facts and is also very ad hoc, and the detection algorithm is very naive.

The tools available for monitoring/debugging distributed programs provide some capabilities to monitor the execution of a distributed program, collect information like the creation of a process, the status of a process, etc., detect complex assertions, and take actions such as halting a computation. The Distributed Programs Monitor (DPM)[5] running on UNIX(1) operating system is a monitoring tool for distributed programs. The DPM consists of a group of cooperating processes which extract primitive events like create/destroy/start/stop-process, creat/destroy-communication-path, and send/wait/receive-message, select only those primitive events which are interesting, and store them. The stored information can be used to analyze the characteristics of a program like the amount of parallelism existing in the program to debug the program using execution replay. But this system collects only the occurrence of a limited set of primitive events and its usage is very much limited. Bates and Wilded[6] and Baiardi et. al.[7] introduce debugging systems in which primitive events like those in DPM are collected, composite events which are written in the regular expression extended with the shuffle operator are detected in real time, and simple actions like halting a computation can be taken. But no treatment of fact type assertions is mentioned. In [8], the attribute values of entities like hosts, processes, etc. and the occurrences of exported operations of those entities are collected and stored in the centralized monitor as a history database. Assertions to be detected are written as queries to the history database. But the query language does not have subqueries and aggregation operators like average, sum, minimum, and maximum. And assertions like "For all the hosts in a certain cluster, the load average is greater than 10" cannot be written. And because this system is a pure monitoring system, there is no concept of actions.

In the DSM that we describe in this paper, we view the managed system as a collection of entities like hosts, processes, files, etc. and an entity is characterized with attributes, exported operations, and reported events. Exported operations and reported events are primitive events. Entities having the same characteristics are grouped into an entity class and among entity classes there exist relationships including the part-of relationship which builds an aggregation hierarchy. Attributes and primitive events are collected using sampling and tracing probes, sent to the DSM, and stored in a history database, which is called Management Information Base (MIB). Assertions can be detected with the data-driven detector and the goal-driven detector. Simple assertions such as "John logs in the machine ernie" can be detected with only the data-driven detector. But the fact like "no report has been received from the machine ernie during the last 30 seconds" can be caused by many different causes such as "ernie is down", "link to ernie is down", or "ernie is very heavily loaded". In this case the fact can be thought as a symptom and the three possible cases can be considered to be three hypotheses. To deal with this kind of situation, the DSM first detects the symptom with the very fast data-driven detector and then decides which is the most probable cause for the symptom using the goal-driven detector which has the capability of reasoning with uncertainty and experiment on remote machines. Therefore the assertion can be written as <symptom, super-

1 UNIX is a Trademark of Bell Laboratories
hypothesis>. All the hypotheses form generalization hierarchy and the super-hypotheses is the lowest level hypothesis which includes all the hypotheses for the symptom. In case an assertion does not need goal-driven detection, it could be written as \(<\text{symptom}, >\). The symptoms are written with a specification language based on classical temporal logic[9,10] and the super-hypothesis is written as a Prolog clause. An action can be a query to the MIB, a notification of the detection of an assertion, or a direct control over the system. An action can be invoked conditionally or unconditionally. If an assertion is not given in a request, the action of that request is taken unconditionally as soon as that request is received by the DSM. Otherwise the action can be taken only after the associated assertion is detected.

In this paper we describe our design decisions on the issues of information collection and the assertion specification language. The rest of the paper is organized as follow. The section 2 describes overall structure of the DSM and the issues related with information collection is discussed in the section 3. The section 4 presents the assertion specification language and is followed by the conclusion in the section 5.

2. OVERALL SYSTEM STRUCTURE

![Diagram of overall system structure]

A simplified picture of the DSM is given in the figure 2.1. There is a centralized DSM managing the whole system which is a collection of homogeneous or heterogeneous nodes. All of these nodes and the DSM are connected by some networks such as a wide area network, a local area network, or a very fast interconnection network.

A DSM consists of several modules and data structures. When a client’s request of the form \(<\text{symptom}, \text{super-hypothesis}, \text{action}>\) is received by the user interface module, its syntactic and semantic correctness is checked, then it is sent to the request manager.

The request manager is the most important module of the DSM and basically it performs three functions. First it turns on or off any required probes in the managed nodes. These probes collect information like the load average of a host, a create-process, a message-send, etc. And this collected information is stored in the Management Information Base (MIB). Second the request manager detects assertions using the data-driven detector, the goal-driven detector, and the information stored in the MIB. The detection algorithm used in the data-driven detector is state-preserving like the RETE algorithm[11,12] and the goal-driven detector is implemented with Prolog extended with interfaces to the user-defined C programs. Third the request manager takes the action in the request using the action handler when the assertion in the same request is detected. When an action is taken, some control packets are sent to one or more managed nodes.

3. INFORMATION COLLECTION

In this section we discuss issues related with the information collection. We present the data-modeling of the managed system based upon the entity-relationship model, the kinds of information that is collected, and the representation scheme with which the collected information is stored.

3.1. Data Modeling of the Managed System

We view the managed system as a collection of various kinds of entities. An entity can be a permanent one like a host or can be a temporary one like a process. An entity is characterized by three lists: attributes, exported operations, and reported events. Attributes represent states of an entity and each of them has a name and a data type. An attribute can be a constant or a time-varying one. Some of the constant attributes can uniquely identify an entity and a minimal set of such attributes is called a key to that entity. An exported operation is an interface function which is provided to the outside. A reported event of an entity is any event that can occur inside that entity but is not an exported operation. Both an exported operation and a reported event are collectively called primitive events and have a list of parameters. For illustration we consider a distributed system with entities such as hosts, accounts, processes, messages, and files. A host has constant attributes machine-name and machine-type and time-varying attributes load-average and status whether it is up or down. In this case the machine name is the key to the host entity. An account has constant attributes login-name and password, a time-varying attribute login-status, exported operations login and logout, and a reported event login-failure. A process has a constant attribute process-id, a time-varying attribute status whether it is running, ready, or blocked, and exported operations create/destroy/start/stop-process. A message has exported operations send/receive-message and a file has exported operations create/destroy/read/write-file. Among the many primitive events, the exported operation
message-send has parameters sender-process-id and receiver-process-id and the reported event login-failure has parameters host-name, login-name, and password.

Entities having the same lists of attributes, exported operations, and reported events are grouped into an entity class. An entity class corresponds to an abstract data type and an entity of that entity class is an instance of that abstract data type. Among entity classes there exist relationships. They are part-of, own, use, and connect. Let's e1, e2, and e3 be entity classes. If an instance of e1 can be properly contained in some instance of e2, then part-of relationship holds from e1 to e2. This part-of relationship defines the aggregation hierarchy of the system. If an instance of e1 can be owned by some instance of e2, then own relationship holds from e2 to e1. If exported operations of an instance of e1 can be invoked by some instance of e2, use relationship holds from e2 to e1. If an instance of e1 is connected to some instance of e2, then connect relationship holds between e1 and e2. Unlike other relationships discussed above this connect relationship is bidirectional. Figure 3.1 is an entity-relationship model of the distributed system that we consider with additional entity classes, cluster and link.

3.2. What Information to Collect

Information is collected by installed probes. A probe can be a sampling probe or a tracing probe. A sampling probe is periodically or aperiodically invoked, reads some attribute value, prepares a data packet with that value, and sends it to the DSM. A tracing probe is invoked when a primitive event occurs, prepares a data packet including information on that primitive event, and sends it to the DSM.

As described in the previous subsection, an attribute can be constant or time-varying. Both a constant attribute and a time-varying attribute can be further subdivided. A constant attribute can be that of a permanent entity or a temporary entity. A constant attribute of a permanent entity (e.g., the machine type of a host) is not collected by any probes but is permanently stored in the MIB in the DSM. A constant attribute of a temporary entity (e.g., the command name of a process) is stored in the MIB when the creation of that entity is reported by a tracing probe and a new slot is prepared for that entity in the MIB. It is assumed that all the constant attribute values of a temporary entity are contained in the data packet reporting the creation of that entity. A time-varying attribute may or may not be associated with the occurrence of a primitive event. If it is not associated with any primitive events (e.g., load average of a host), its value is collected by a sampling probe. A data packet generated by a sampling probe has the entity identification, the attribute name, the attribute value, and the time when the attribute value was sampled. But some time-varying attributes can have their values changed only when there occurs a primitive event which affects the value of that attribute values. For example, the process status which can be running, blocked, ready, or etc. can change when a process is selected from a ready queue and made to run, a process is blocked to wait for an input from an input/output device, or a running process is stopped after the time-out and put into the ready queue. So whenever that kind of primitive events are reported by tracing probes, the time-varying attribute is updated in the MIB.

Occurrences of primitive events are reported by tracing probes. The data packet generated by a tracing probe has all the parameter values and the start and end times of the occurred primitive event. Information on relationships among entities need not be collected because they can be either permanently stored in the MIB or constructed from primitive event reports.

3.3. Storage Representation

In this subsection we describe the representation scheme in which the collected information is stored in the MIB. All the information collected by probes is temporally qualified, i.e., has time-stamps. The value of a time-varying attribute has a time point from which that value becomes valid. A primitive event also has a time interval during which that event occurred.\[13\] introduces three approaches to represent these kinds of temporal information. They are state-space approach, date-line approach, and temporal-graph approach. These approaches are used in simple problem-solving tasks\[14\], temporal database systems\[15, 8, 16\], and natural language representation\[9, 17\] respectively.

The date-line approach is chosen in the DSM because this is the most efficient approach to our case where all the information reported by the sampling and tracing probes has explicit temporal values with it. Then we have to decide the characteristic of the structure of time points which are associated with the collected information. The structure of time points can be either discrete or continuous. If it is discrete, there exists a real number \( \varepsilon > 0 \), such that the distance between any two points is at least \( \varepsilon \). Otherwise it is continuous. And the structure of time points can be linear or branching. In the linear structure, all the time points are totally ordered so all of them can be mapped to a real line. In the branching structure, a line to which time points are mapped can be divided into several lines and some of them can merge later. In the computer systems that we are dealing with, there exists a minimal time unit (e.g.,

![Figure 3.1. Entity-Relationship model of the System](image)
microsecond, millisecond, etc.) with which we can measure our measurements varying. A time-varying attribute is stored time linear attribute values or primitive events. And all the two neighboring time points is the minimal time unit available from the measurement.

In the MIB all the information on an entity class, a primitive event type, or a relationship type is stored in one table (or a relation in the relation database terminology) respectively. In figure 3.2, tables for the host entity, the message-send event, and the connect relationship are presented. The host entity has four attributes among which the name and the machine-type are constant and the load-average and the status are time-varying. A time-varying attribute is stored as a collection of \((t_i \rightarrow v_i)\), meaning that the attribute has value \(v_i\) during the time interval \([t_i, t_{i+1})\). This interval is closed at \(t_i\) and open at \(t_{i+1}\). The null value means that the attribute value is not defined during the time interval. The message-send primitive event has two attributes, sender-pid and receiver-pid. The first columns in the message-send primitive event table and the connect relationship table have no name and are the lifespan of the tuple, i.e., the set of time intervals during which the corresponding tuple is valid. In the host entity table this column is omitted because the host is a permanent entity and therefore the lifespan is infinite. And if the interval has zero-length then it is written as a single time value. The message-send primitive event table shows that the process 1450 sent a message to the process 980 during [3.3] or at 3 and the process 980 sent a message to the process 1450 at 5. The connect relationship table shows that the host ernie has been connected to the link Link-1 since 1 and the host arpa was connected during [2.4] and [6.8].

4. ASSERTION SPECIFICATION

In this section we present a specification language for assertions. Because the super-hypothesis of the assertion can be written as a Prolog clause, in this section we just consider an assertion without super-hypothesis. So the assertion here corresponds to the symptom. To be able to represent the temporal relations among component assertions we choose the temporal logic as the basis for the specification language. Suppose we want to represent the assertion that a certain host ernie is up at time \(t\). Some of the questions that can be asked when we write this temporally qualified sentence in a logical form are[10]

1. Over what do we interpret assertions: The question here is what \(t\) represent. It can be a point, a time interval, or both of them. Each case is called the point-based, interval-based, and mixed temporal logic. In our system, a system property can hold and an event can occur at a time point or during a time interval. Also a time-point can be represented as a special case of time-interval where two end points are the same. Therefore we choose the interval-based approach. The mixed approach is eliminated because it does not give a uniform way to associate time.

2. What are the primitive temporal objects: The question is what is the most primitive element of the time structure. Most of the time it is either a point or an interval. If a time point is chosen to be the primitive temporal object, then an interval of the interval-based or mixed temporal logic can be represented with two time points, one for the start time and the other for the end time. The only temporal relation between two time points is \(r\ precedes r'\). If the primitive temporal object is the time interval, we can have only the interval-based logic because the representation of a time point in terms of time intervals is complex, although not impossible. Between two intervals there can exist many kinds of relations like equal, before, overlap, during, etc. Because we want to represent both time points and time intervals efficiently we choose the time-point as the primitive temporal object.

2. What logical form: We also have to choose the actual method for associating an assertion with a time point or interval. One method is to separate the temporal and atemporal components of an assertion, introduce a global predicate like \(TRUE\), and let both the temporal and atemporal components of the assertion be the arguments of this global predicate. This approach is called the classical temporal logic. In this case we have \(TRUE\, t\), \((up\ ernie\)). Another method is to follow the modal logic where the atemporal component of an assertion is associated with modal operators like always, sometimes, and next. In this case the time is not explicitly mentioned at all. Because we need explicit representation of the temporal components we take the first approach.

In this paper we use all the logical connectives in its standard interpretation with following syntax.

\[(and\ fmla_1\ fmla_2)\]
\[(or\ fmla_1\ fmla_2)\]
\[(not\ fmla)\]
\[(forall\ (vars)\ fmla)\]
A temporally qualified assertion is written as

\[ (\text{TRUE } t_1 \leq t_2 \text{ nontemporal-assertion condition}) \]

This assertion states that a nontemporal-assertion is true between \( t_1 \) and \( t_2 \) and also satisfies the condition. \( t_1 \) and \( t_2 \) are the starting and ending times of the nontemporal-assertion respectively and their syntax is given in the figure 4.1. A nontemporal-assertion can either be a fact or an event. A fact is a system property which holds over an interval. If the nontemporal-assertion is a fact \( f \), then we have

\[ (\text{if } (\text{TRUE } t_1 \leq t_2 f \text{ cond}) \)
\[ (\forall t (\text{if } (\text{TRUE } t_1 \leq t \leq t_2) (\text{TRUE } t f \text{ cond}))))) \]

That is, if a fact holds during an interval \( I \), then it holds in any subinterval of \( I \). But if the nontemporal-assertion is an event \( e \), then

\[ (\text{if } (\text{TRUE } t_1 \leq t_2 e \text{ cond}) \)
\[ (\forall t (\text{if } (\text{TRUE } t_1 \leq t \leq t_2) (\text{TRUE } t e \text{ cond}))))) \]

That is, if an event occurred during an interval \( I \), then there is no subinterval of \( I \) over which the event occurred. The condition is the constraint on the variables appearing in the nontemporal-assertion, is written as a boolean expression of those variables and constants, and can be omitted if there is no constraint.

<table>
<thead>
<tr>
<th>( t_1 )</th>
<th>( t_2 )</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant</td>
<td>constant</td>
<td>fixed starting time and fixed duration</td>
</tr>
<tr>
<td>constant</td>
<td>variable</td>
<td>fixed starting time and variable duration</td>
</tr>
<tr>
<td>variable</td>
<td>constant</td>
<td>fixed ending time and variable duration</td>
</tr>
<tr>
<td>variable</td>
<td>variable</td>
<td>variable starting time and variable duration</td>
</tr>
<tr>
<td>variable</td>
<td>(+\text{constant})</td>
<td>variable starting time and fixed duration</td>
</tr>
<tr>
<td>(+\text{constant})</td>
<td>(+\text{constant})</td>
<td>(two variables should be the same)</td>
</tr>
</tbody>
</table>

Figure 4.1. Syntax for the starting time and the ending time

Now we present the syntax of the assertion \( \text{ast} \) as follow.

\[ \text{ast} := (\text{TRUE } st \text{ et nontemporal-assertion condition}) \]
\[ (\text{AND } \text{ast} \text{ ast condition}) \]

The semantics of \( \text{AND} \) and \( \forall \) are

\[ (\text{if } (\text{AND } \text{ast1} \text{ ast2 condition}) \)
\[ (\forall \text{ast1} \text{ ast2 condition}) \]
\[ (\text{if } (\forall \text{ast1} \text{ ast2 condition}) \)
\[ (\text{forall} \text{ ast1} \text{ ast2 condition})) \]

The \( \text{OR} \) operator has the same meaning as \( \lor \) except the constraint that two \( \text{ast} \)'s in the expression \( (\text{OR } \text{ast ast}) \) should have the same set of variables. The \( \exists \) operator has exactly the same meaning as the logical connective \( \exists \).

Now we present the syntax for the nontemporal assertions. A nontemporal assertion is either a fact or an event. A fact is built from entity attributes and constants to assert that a certain relation (usually greater than, smaller, etc) holds among them. It has the following syntax.

\[ \text{fact} := (\text{FOR} \text{ fact fact}) \]
\[ (\text{IF fact fact}) \]
\[ (\text{forll } (\text{relation-name entity-name entity-name}) \]
\[ (\text{expr expr}) \]
\[ (\text{term term}) \]

A fact can be a primitive fact or any number of primitive facts which are connected by \( \text{FOR} \) operator. The semantics of \( \text{FOR} \) operator is as follow.

\[ (\text{if } (\text{TRUE } t_1 \leq t_2 (\text{FOR} \text{ fact1 fact2})) \]
\[ (\text{forall} (t) (\text{if } (t_1 \leq t \leq t_2) (\text{TRUE } t t f\text{ fact1 fact2}))) \]

A primitive fact can be a comparison of two expressions which can be a constant, an attribute value at the current time or some previous time, statistics of a time-varying attribute value during some interval, statistics of attribute values of all the entities satisfying a certain condition, or any arithmetic combination of them. A primitive fact can also be built from a relationship to
assert that a certain relationship exists between two entities.
Some examples of facts are

1. (> (host ernie load-avg) 10.0)
2. (> (avg (host ernie load-avg) 10) 1.0)
3. (connect ernie X)

The first example states that the load average of the host ernie is greater than 10, the second that the average of ernie's load average during the last 10 time units is greater than 10, and the third that the link denoted as a variable X is connected to ernie.

A composite fact can be constructed from these simple facts and operators like FAND. So we can have a composite fact like

(FAND (> (host X load-avg) 10.0) (connect X Link-1))

meaning that some host denoted as a variable X has the load average greater than 10 and is connected to the link Link-1. This operator is just syntactic sugar and has semantics as follow:

(if (TRUE tl t2 (FAND fact1 fact2))
 (AND (TRUE tl t2 fact1) (TRUE tl t2 fact2)))

An event can be built from primitive events with the following syntax.

(primitive-event-name parameter-list)

Some examples are

1. (login ernie john)
2. (message-send 1340 908)

The first example states that John logs in the host ernie and the second that the process 1340 sends a message to the process 908.

Like facts, a composite event can be constructed from simple events and operators like AND, OR, and NEXT. All of these operators are syntactic sugar and have the following semantics.

(if (TRUE tl t2 (EAND event1 event2))
 (EXISTS (t1 t2 t12 t21 t22)
 (AND (TRUE t1 t1 t2 event1)
 (TRUE t2 t1 t22 event2)
 (and (t1 = min (t11 t12))
 (t2 = max (t12 t22)))))

(if (TRUE tl t2 (EOR event1 event2))
 (OR (TRUE tl t2 event1) (TRUE tl t2 event2))

(AND (TRUE tl t3 event1)
 (TRUE t4 (t2 event2) (t3 ≤ t4))))

With all the constructs described above some examples of the assertions are:

1. (EXISTS t)
   (AND (TRUE t t (login ernie john))
   (TRUE t +30
   (> (host ernie load-avg) 5.0)))))
2. (EXISTS (X)
   (FORALL (X) (TRUE t t (part-of X planet))
   (TRUE t t
   (> (host X load-avg) 10.0)))))

The first assertion states that at time t John logs in the machine ernie and then during the next 30 time units the load average of the host ernie is not lower than 5. The second assertion says that at time t for all the host in the cluster planet, the load average is not lower than 10.

5. CONCLUSION

In this paper we described the Distributed System Manager being developed on Sun microsystems workstations connected by an Ethernet at University of California, Berkeley. It monitors the distributed system, collects information, detects assertions, and takes actions. Although for the purpose of the clarity of presentation all the discussion is restricted to a distributed computation environment consisting of entities like hosts, processes, messages, etc., the underlying basic idea is general. We believe that the paradigms that we introduced in this paper can be applied to many other areas like the management of manufacturing facilities, the battle management, etc.

References

7. F. Baiardi, N. D. Francesco, and G. Vaglini, "Develop-


